

# DEEP DIVE TO NATURAL LANGUAGE PROCESSING

## OBJECTIVE :-

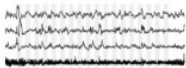

- ✓ An understanding of the effective modern methods for deep learning
- ✓ Basics first, then key methods used in NLP : Recurrent Networks, Attention Based Models
- ✓ A big picture understanding of human languages and the difficulties in understanding and producing them
- ✓ Word meaning, dependency parsing, machine translation, question answering
- ✓ Understand the philosophical foundation of Natural Language Processing
- ✓ Understand sequence models
- ✓ Understand the application of Natural Language Processing
- ✓ Understand the implementation of Natural Language Processing with PyTorch

## SEQUENCE MODELS

There are some problems that are in a sequence form, existing one after the other over a period of time.

Sequential data means whenever the points in the dataset are dependent on the other points in the dataset the data is said to be in a sequential form.

### What is a sequence?

- "This morning I took the dog for a walk." sentence
-  medical signals
-  speech waveform

A sequence could be like a sentence "This morning I took the dog for a walk", this is one sentence but it consists of multiple words and the words depend on each other.

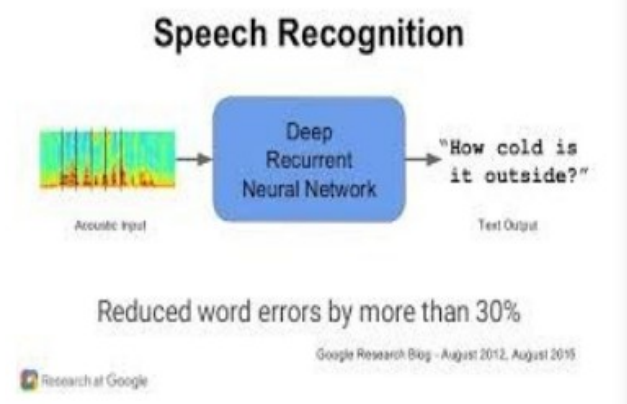
Another example would be like a medical record, one medical record would be one example but it consists of many measurements.

Another example would be like a speech wave from where this one waveform is one example but again it consists of many measurements.

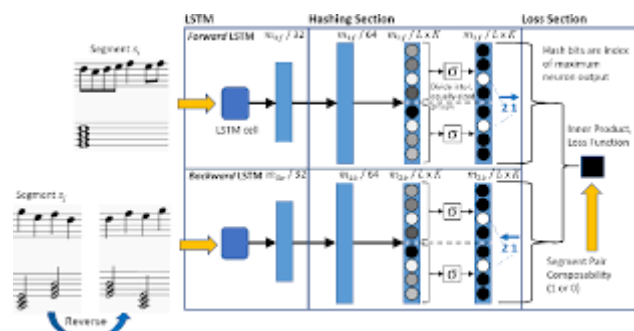
Some of the problems of sequence model

**1) Speech Recognition :-** Given the input  $x$  which is the speech and then translated into a word  $y$  as an output. The speech is a sequence of audio clip plays over time. In these cases both the input and the output are a sequence

data, the input is a sequence of input over time and the output  $y$  is a sequence of words.

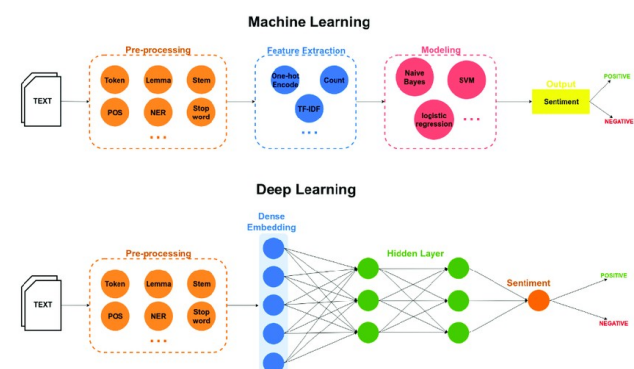


**2) Music Generation :-** This is another problem of a sequence model, the input  $x$  is not a sequence model, it is an empty set ( $\emptyset$ ) or a list of notes and the output is a generated music.



**3) SENTIMENT CLASSIFICATION :-** the word sentiment means view or opinion. Sentiment analysis is a powerful text analysis

tool that automatically mines unstructured data (social media, emails, customer feedback and more) for opinion and emotion and then classifies it with one of the labels that we provide. For example, you could analyze a tweet to determine whether it is positive or negative, or analyze an email to determine whether it is happy, frustrated or sad or if you are a hotel owner, the customers will give you a text-based feedback then the algorithm will classify it into rates, how many stars these reviews would be?



**4) DNA SEQUENCE ANALYSIS :-** This is also another problem of a sequence problem. The input X is a DNA sequence and the output y, which says, which part of these DNA sequence corresponds to a protein.

**Example:-**

Input-X:- AGCCCCTGTGAGGAAGTAG

Output:-y:- AG**CCCCTGTGAGGA**ACTAG

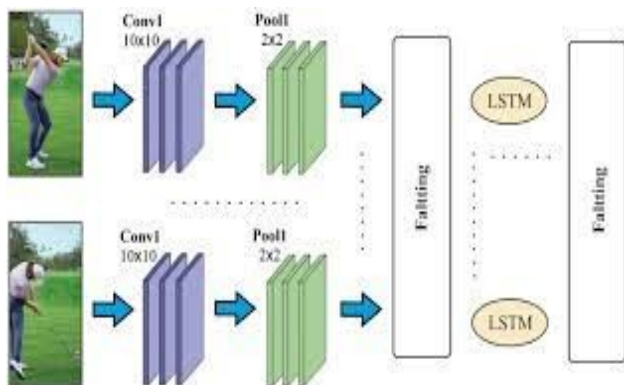
DNA is represented using alphabet A, C, G. In the output the red part of the DNA refers to protein.

**5) MACHINE TRANSLATION :-** this is another sequence problem where you input a sentence and then expect translation in another language.

Example :- French Input :- Voulez vous chanter avec moi

English output :- Do you want to sing with me ?

**6) VIDEO ACTIVITY RECOGNITION :-** We will have a sequence of video frames as input and then the output y is to recognize the activity.



**7) NAME ENTITY RECOGNITION :-** We will give a sequence of sentence as input X and ask the sentence the name of the people in that sentence

**Example :-**

Input X :- Yesterday Harry potter met Hermione Granger

Output y :- Yesterday **Harry potter** met **Hermione Granger**

Why do we need name entity recognition ? because sometimes we may need to index (to index means to organize them in a certain schema) all the people mentioned in the last 24 hours of news, articles but not only people's name but also company names, location names, country names, currency names (Birr, dollar, pound).

So all of these supervised machine learning problems were there is mapping between an input x and a label y. So in these problems may be both of the input x and the output y are sequences or one of them might be sequences

and one might not. sometimes the length of the input sequence and the output sequence might be equal or sometimes not.

## NOTATIONS USED TO BUILD SEQUENCE MODELS

X :- Harry potter and Hermione Granger invented new spell

y :- 1 1 0 1 1 0 0 0

We have one output per input word and the target output to the desire y and tells you for each of the input words is the part of the person's name.

X :- (Harry potter) and (Hermione Granger) invented new spell.

y :- 1 1 1 0 1 1 0 0 0

There is another sophisticated way of representation which tells you not only is a word is a part of a person's name but tells you where the start and the end of people's names in the sentence.

So the input is a sequence of nine words, so we are going to have nine features to represent this nine words and index in to the position and the sequence as a super script for each of the features.

X :-  $x^{<1>}$  Harry  $x^{<2>}$  potter  $x^{<3>}$  and  $x^{<4>}$  Hermione  $x^{<5>}$  Granger  
 $x^{<6>}$  invented  $x^{<7>}$  new  $x^{<8>}$  spell

y :- 1 1 0 1 1 0 0 0  
 $y^{<1>}$   $y^{<2>}$   $y^{<3>}$   $y^{<4>}$   $y^{<5>}$   $y^{<6>}$   $y^{<7>}$   $y^{<8>}$ .

small t represents the time step. What does it mean by time steps ? In Natural Language Processing (NLP), a time step may be associated with a character, a word or a sentence, depending on the set up.

To know the length of the words in the input sequence  $T_x = 8$ , to know the length of the words in the output sequence we will be represented as  $T_y = 8$ . In this case (in the name entity recognition problem) both the inputs and the output length is equal.

In the notation that we saw  $x^{(i)}$  to denote the  $i^{\text{th}}$  training example, so to refer the  $t^{\text{th}}$  element of the sequence of the training example i, will use this notation  $x^{(i)<t>}$

and if  $T_x$  is the length of the sequence then  $T_x^{(i)}$  is the length of the sequence in that training example.

Similarly  $y^{(i)<t>}$  means the  $t^{\text{th}}$  element of the sequence of the output label for the  $i^{\text{th}}$  sample and if  $T_y$  is the length of the sequence then  $T_y^{(i)}$  is the length of the sequence in that  $i^{\text{th}}$  sample.

## HOW TO REPRESENT EACH WORDS IN THE SEQUENCE (HOW TO CONVERT THE PROBLEM IN A DIGITALIZED FORM ) ?

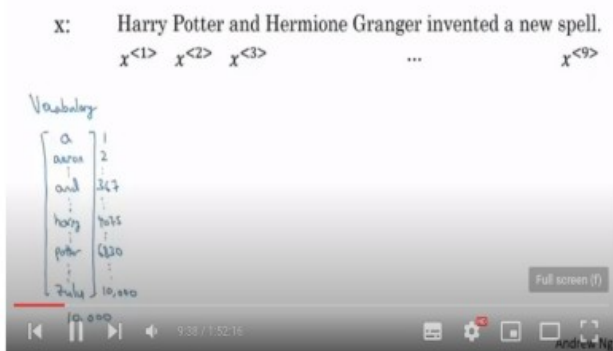
### REPRESENTING THE WORDS

X :- Harry    potter    and    Hermione    Granger  
                x<sup><6></sup>    x<sup><7></sup>    x<sup><8></sup>  
                invented    new    spell

We are now working on a Natural Language Processing and one thing that we need to decide is how to represent individual words in the sequence , like how do we represent the word harry and what should X<sup><1></sup> really be.

So to represent (changing a certain problem in to a number ) the words in the sentence , the first thing we do is come up with a vocabulary or sometimes dictionary that means making a list of the words that you will use in a representation.

### Representing words



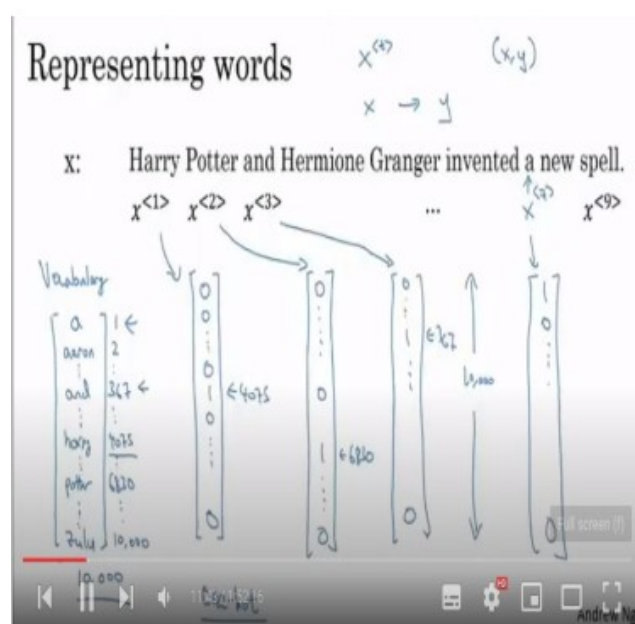
The dictionary starts with “a” and the number in the left shows the indexes or the position of the words in the vocabulary.

In this example , we are using a dictionary with the size of 10,000 words this is quite small by the modern NLP applications. For

commercial applications dictionary size of 30-50 thousand words are more common and 100 thousand is not uncommon and some the large internet companies will use a dictionary size of a million words may be bigger than that.

So after deciding the size of our dictionary , one way to build these vocabulary would be to look through your training set and find the top 10,000 occurring words or to look through some of the online dictionaries that tells you what are the most common 10,000 words in the english.

So in order to represent the words in our training corpus , which means in order to change our problem in to numbers , in order to be computed by our algorithm , one way to do is through one hot encoding.



We are doing these using supervised machine learning by giving a pair of (x,y) , what if we encounter a word that is not in our vocabulary , well the answer is we create a new token or a new fake word called “Unknown word” to represent words that are not in our vocabulary.

### HOW DO WE REPRESENT THE MEANING OF A WORD ?

**What does it mean by meaning :-** The idea that is represented by a word , phrase , it’s the idea that a person wants to express using words , signs etc.

**Representing words as discrete symbols :-** In traditional NLP (basically we call this for the years before 2012 , before using neural nets) we regard word as discrete symbols , means one 1 and the rest 0’s , from the dictionary(words can be represented by one hot vectors)

**Example :-**

motel = [0 0 0 0 0 0 0 0 1 0 0]

hotel = [0 0 0 0 1 0 0 0 0 0 0 0]

Language have a lot of words and one of those dictionaries that you had probably have about 25,000 words in them , but if we start getting more technical and scientific english it’s easy to get to a million words.

Actually the number of words that we have in language like english is infinite because we have these processes which are called derivational morphology(structure). where we can make more words by adding endings on to existing words. So we can start with something like paternalist and then we can say from paternal to paternalist or paternalistic , paternalism , paternalistically and so we really will end up in to an infinite space of words.

## LIMITATION OF ONE HOT ENCODING

So that is a minor problem, having a big vectors if we want to represent a sensible size of vocabulary. The much bigger problem, which is precisely what we want to do all the time is we want to sort of understand relationships and the meaning of words. So obvious example of this is a web search, so if we do research for Seattle motel, it will be useful if also showed me results that had Seattle hotel on the page and vice versa, because hotel and motel are actually the same thing but if we have these one hot vectors like we had before they have no similarity relationship between them, in math terms these two one hot vectors are orthogonal, no similarity or relationship among them (They are perpendicular to each other and their dot product is zero)

## WHAT IS WORD EMBEDDING ?

Before we get in to word2vec, let's establish an understanding of what word embeddings are, this is important to know because the over all result and output of word2vec will be embeddings associated with each unique word passed through the algorithm.

Word embedding is a technique where individual words are transformed in to a numerical representation of the word, where each word is mapped to one vector, this vector is learned in a way which resembles a neural network. The vectors try to capture the various characteristics of that word with regard to the overall text. These characteristics can include semantic relationship of the word, definitions, context etc with these numerical representations, you can do many things like identify similarity or dissimilarity between words.

Questions :-

- 1) How do we transform the words in to a numerical representation
- 2) How do we teach this vectors to resemble a neural network and extract a good characteristics out of it.

A machine can not process text in its raw form, thus converting the text in to embedding will allow users to feed the embedding to classic machine learning models. The simplest embedding would be a one hot encoding of text data where each vector would be mapped in to a category but as we know one hot encoding has a problem, they do not capture characteristics of the word and they can quite large on the size of the corpus.

For example :- let's say we have a language model that has learned, "I want a glass of orange \_\_\_\_", so very likely to be juice, so even if the learning algorithm has learned that i

want a glass of orange juice is a likely sentence but if we say :- I want a glass of Apple \_\_\_\_, as far it knows the relationship between Apple and orange is not any closer as a relationship between any of the other words, like man-woman, king-queen and orange. The relationship

between the word orange and the word apple is no any closer in the vocabulary, orange is indexed in the position 6357 and Apple is indexed in the position 456. so this means it just doesn't know that somehow Apple and orange are much more similar than kind and orange or queen and orange. So if we represent one hot encoding, it tries to learn the relationships by the index in the vocabulary.

## FEATURIZED REPRESENTATION OF WORD EMBEDDING

one-hot position of 5372

	Man (2311)	woman (2352)	King (4314)	Queen (7109)	Apple (446)	Orange (6237)
Gender	-1	1	-0.95	0.97	0.00	0.01
Royal	0.01	0.02	0.93	0.95	-0.01	0.01
Age	0.03	0.02	0.7	0.69	0.05	-0.02
Food	0.04	0.01	0.02	0.01	0.95	0.97

size  
- core  
- alive  
- action  
- noun  
- verb

→ We can imagine food coming out with many features

Man & woman doesn't tell anything about age

King and Queen are almost study

Apple

REDMI NOTE 6 PRO MI DUAL CAMERA

So the numbers in the featurized implementation shows that the effect of that feature on that specific word in the vocabulary, like how the word "man" is affected by the feature "food", so as we know we are converting our problem in to numbers so that it will be easy for the algorithm to understand this numbers and extract a pattern out of it, the lower number shows that this feature has almost no effect to that word and a high word shows that this feature has effect to that word.

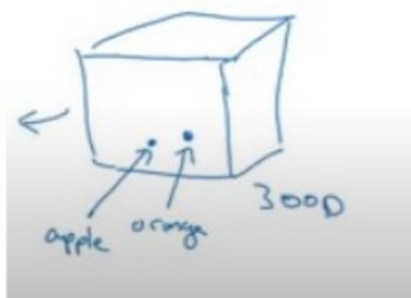
So for now, let's say 300 different features and what that does is it allows us to take is a list of 300 dimensional vector so each word will have this 300 dimensional vector. If we use this representation to represent the words orange and Apple, then notice that the representation for orange and apple is quite similar. This means most of the features not all of the features, some of the features may be differ because may be the color of orange and the color of apple, the taste order or some feature of apple and orange are actually the same, so this increases the odds of the algorithm that has figured out that orange juice is a thing to also quickly figure out apple juice is also a thing. So this representation helps us to generalize better across different words.

**So what is the use of word embedding ?** It helps us to generalize different words and understand the meaning of the word by embedding it in to so many features. So we will find a way to learn a word embeddings which is basically seem to learn high dimensional feature vectors like this, that gives a better representation that one hot



vectors. So featurized representing will allow an algorithm to quickly figure out that Apple and Orange are more similar than king and orange or queen and orange.

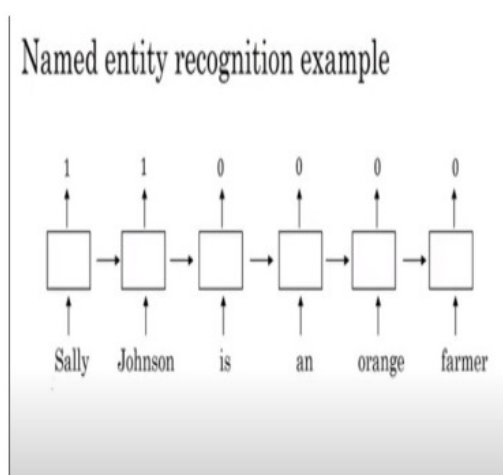
This algorithm will find words like man and woman will be grouped together, fruits will be grouped to each other and number will be grouped together, humans and animals can be grouped as animate objects and these Featurized representation, may be 300 dimensional space, these are called embeddings and the reason we called them embeddings is we can think each and every word and get embedded to a certain point in these 300 dim space. Example :- the word orange is embedded like in the image and also the word apple is embedded in this 300 dimensional space.



### Pre-Trained Word Embedding

Now let's take these featurized representation and plug them in to NLP applications.

If we are trying to detect people's name given a sentence like "sally Johnson is an orange farmer",



hopefully we figure out that "sally Johnson" is a person's name, the output  $y$  is one and one way to be sure that sally Johnson is a person's name rather than it's a corporation name is that "orange farmer" is a person. Previously we've seen one-hot representation to represent  $x^{<1>}$ ,  $x^{<2>}$ ,  $x^{<3>}$ , ...,  $x^{<T>}$  but if we can now use a featurized representation, the embedding vectors after we

trained a model that uses word embedding as the inputs, if we now see a new input in the test set :- "Robin Linn is an apple farmer", so from the training knowing the orange and the apple are very similar, so it will make it easier for our learning algorithm to generalize that "Robin Linn" is also a person's name.

But what if in our test set, we see not Robin Lin is an apple farmer but we much less common words, what if we see "Robin Lin is a durian cultivator", where durian is a rare kinda fruit which mainly exists in Singapore. But if we have a small training set for the named entity recognition task you might not even see the durian or cultivator in our training set, but if we learned a word embedding that tells you durian is a fruit and cultivator is someone cultivates is a farmer then we might still can generalize from having seen an orange farmer in our training set to knowing that durian cultivator is also a person.

So one of the reasons where the embeddings will be able to do this is the algorithm for learning word embeddings can examine very large text corpses may be found off the internet so we can examine very large datasets may be a million words, may be even up to 100 billion words this could be quite reasonable so very large training sets are just unlabeled text and by examining tons of unlabeled text which you can download for free, so that you can figure out that durian and orange are kinda similar and therefore learn embeddings that can group them together.

So we discovered that orange and durian are both fruits by reading massive amount of internet text, what we can do is take this word embedding and apply in to our named entity recognition system, for which we might have a much smaller training set may be even just a hundred thousand words in our training set or even much smaller, so this allows you to carry our transfer learning, where we take information we learned from a huge amount of unlabeled text, essentially from the internet to figure out that orange, apple and durian are fruits and then transfer that knowledge to a task such as named entity recognition for which we might have relatively small labeled training set.

### Transfer Learning and Word Embedding

1) Learn word embeddings from large text corpus (1-100B words) or we can download pretrained embedding online under permissive licenses.

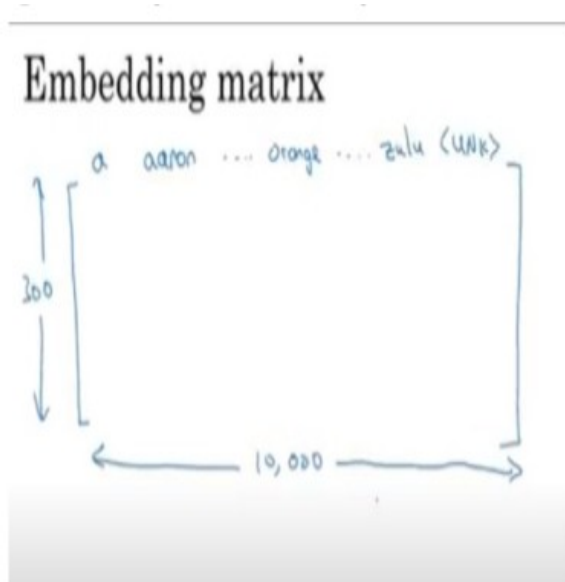
2) And we can take those words and embeddings to transfer embedding to new task with smaller training set (say 100k words)

3) Optional :- Continue to fine tune the word embeddings with new data, optionally we can continue to fine tune to adjust the word embeddings with the new data, on practice we would do this only if our task (number-2) has pretty big dataset. we will fine tune if our task has larger dataset then we will fine tune it (may be train earlier layers)

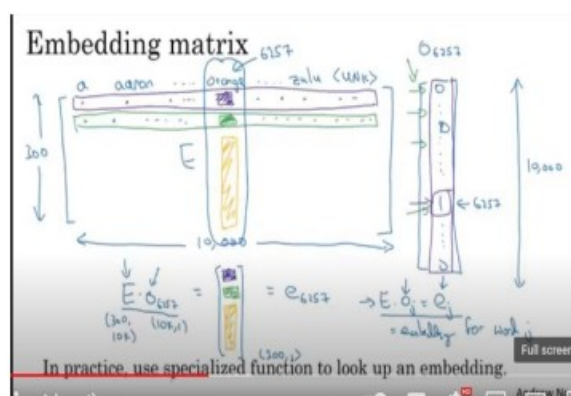
So transfer from some task A to some task B , the process of transfer learning is most useful when you happen to have a ton of data for A and relatively smaller dataset for B and that is true for a lot of NLP tasks.

### Embedding Matrix

When we implement an algorithm to learn a word embedding what we end up learning is embedding matrix.



300 \* 10,000 word in vocabulary , adding the unknown word 10,001, one extra token



And the columns of these matrix would be different embeddings for the 10,000 different words we have in our vocabulary . So orange was word number 6257 in our vocabulary of 10,000 words , so one piece of notation we will use is that  $O_{6257}$  was the one hot vector , with zeros ever where and a 1 in position 6257 and it will be a 10,000 dimensional vector with a 1 in one position.

So if we called the Embedding matrix  $E$  , the notice that , if we take  $E$  and multiply it by  $O_{6257}$

$$\begin{matrix} \downarrow \\ E \cdot O_{6257} \\ (300, \quad 100) \quad (10000, 1) \end{matrix} = \begin{bmatrix} \text{...} \\ \text{...} \\ \text{...} \end{bmatrix} = e_{6257}$$

And notice that to compute the first element of this 300 dimensional vector , multiply the first row of  $E$  with the  $O_{6257}$  and end up like in the image above and all of these elements in the  $E$  will be zero except the element  $E_{6257}$ .

$$E \cdot O_{6257} = e_{6257} \rightarrow (300, 1)$$

So  $e$  for a certain specific word  $w$  will be  $e_w$  - the embedding for the word  $w$  , so more generally  $E \cdot O_j = e_j$

$O_j$  :- One hot vector with a 1 at the position  $j$  , where  $e_j$  is the embedding vector for the word  $j$  in the vocabulary. Our goal is to learn an embedding matrix  $E$  but when we are implementing this it's not efficient to actually implement this as a matrix vector multiplication , because the one hot vectors are relatively high dimensional vector and most of the elements are zero , so it's not efficient to use a matrix vector implementation.

So in practice we will actually use a specialized function that just look up a column of the matrix "e" rather than do this with the matrix multiplication but when we write it out in math its convenient to write out  $E \cdot O_j = e_j$

In keras , there is an embedding layer , then it more efficiently pulls out the column from the Embedding matrix  $E$  rather than does it with a slower matrix vector multiplication.

### LEARNING WORD EMBEDDING

"You shall know a word by the company it keeps" , This is a famous quote by British linguist John Rupert Firth -- He is popularly known for drawing attention to the fact that you can tell the meaning of a word by looking at other words in the same context in any given sentence. This means that words that can be used interchangeably in a sentence share similar meanings.

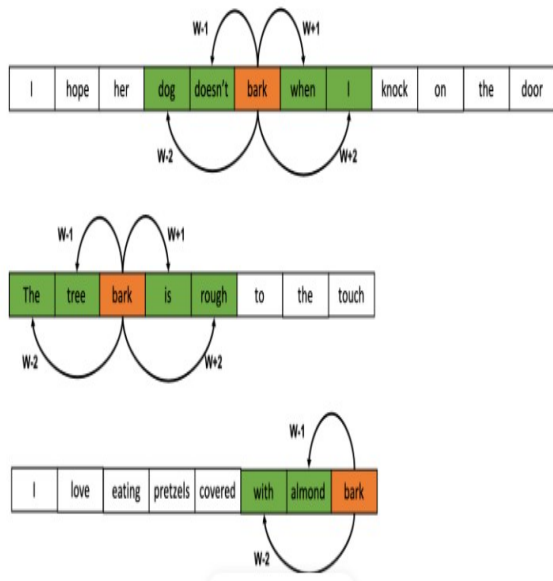
The idea of "Context-dependent" meaning of words makes more sense when you look at homonyms. There are words that have different meanings in a different contexts. An example is "bark" in the sentences below

- I hope her dog doesn't bark when I knock on the door
- The tree bark is rough to the touch
- I love eating pretzels covered with almond bark

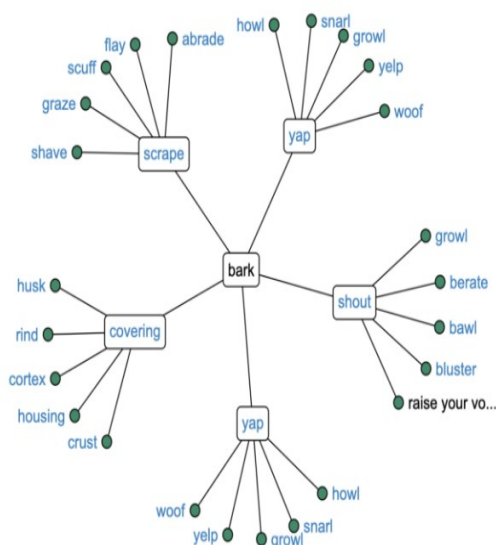
In all three sentences , you can immediately tell the meaning of bark by looking at the words around it. In

practice you would usually select a window of surrounding words and try to infer the original word's meaning by looking at the selected words known as "context words" and we would refer to "bark" as the center word.

Question :- If one word can be represented according to the contexts , does this mean we are going to generate so many vectorized implementation for just one single word , to understand the meaning.



The image below shows the synonyms of the word "bark" in different contexts. This means in the first sentence , we can easily replace the word "bark" with "snarl" , "growl" but not completely with "crust" because they mean completely different things despite being a synonym of bark.



Therefore when modeling words in to vectors , it is important to encode them in a manner that reflects their meaning in the contexts that they appear and this is the intuition behind the word2vec algorithm.

Word2vec is an NLP algorithm that encodes the meaning of words in to short , dense vector (word embeddings) that can be used for downstream NLP tasks such as Question Answering , Machine Translation , Language Modeling e.t.c

Question :-

- How does Word2Vec will encode the meaning of words in to short and dense vector ?
- Do we have different vector representation for the same word in different contexts ?

Before word2vec was introduced words were represented as sparse long vectors with dimensions the size of the entire vocabulary present in the training corpus. Examples of these traditional vectors include one hot vectors. One of the major drawbacks of representing words as sparse vectors is establishing any form of relationship between words. This is because these vectors do not contain enough information about the words to demonstrate such syntactic or semantic relationships Example one hot vectors are orthogonal (Perpendicular and have a dot product of zero) and thus can not be used to measure any form of similarity.

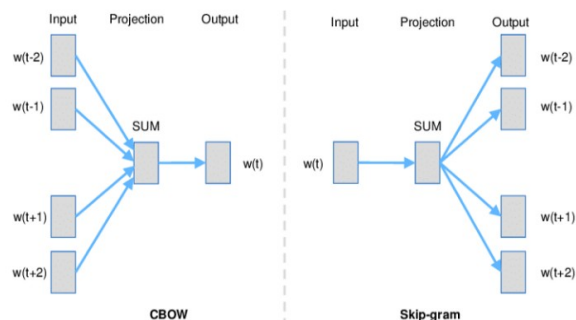
## HOW DOES WORD2VEC SOLVES THIS PROBLEM

As mentioned earlier , the intuition behind word2vec is to ensure that words that exist in similar contexts in sentences are mapped to the same vector space .This means that words with similar neighboring/surrounding/context words in a corpus have similar vectors (with high cosine similarity). This means we can use these words interchangeably.

Even more impressive is that the similarity of word embeddings goes beyond syntactic regularities , using simple algebraic expressions , we can show more complex relationships between words For Example :- The authors were able to establish the vector ("King") - vector ("man") + vector (women) results in a vector with similarity closest to the vector representation of "Queen" Vector("Queen")

Question :- How do we know two words have similar context ?

So word2vec is a technique for Natural Language Processing published in 2013. The word2vec algorithm uses a neural network model to learn word associations from a large corpus of text.



Word2vec is a two layer neural network that processes text by “Vectorizing” words. Its input is a text corpus and its output is a set of vectors. Feature vectors (short and dense vector) that represent words in that corpus. Once trained such a model it can detect synonym words or suggest additional words for a partial sentence. As the name implies word2vec represents each distinct word with a particular list of numbers called a vector. The vectors are chosen carefully such that a simple mathematical function (the cosine similarity between the vectors) indicates the level of semantic similarity between the word represented by those vectors.

## ENCODING WORDS IN THE CONTEXT THEY ARE APPEARING

There are two architectures used for training word2vec embeddings.

- 1) Continuous Bag Of Words (CBOW)
- 2) SKIP GRAM

**Continuous Bag Of Words :-** learns word representations by predicting a center word from a window of selected context words. In CBOW , we sample a window of context words around a particular word , feed it to the model and predict the center word , in this particular architecture the weight matrix between the input and the projection layer is shared between all the words. we map one hot vectors of the input vectors (context words) on the projection layer (embedding layer). The embedding layer of n-dimension is multiplied by another weight matrix to get the output layer. Finally we will run a Softmax operation on the output layer to get probability distribution over the words in the vocabulary.

**Skip Gram :-** is a different variant of word2vec , unlike CBOW , where we predict a center word based on context words in a vocabulary. Here we are trying to learn word vector representation by predicting the context words around a particular word. The model tries to maximize the classification of a word based on another word in the sentence. Ideally , the longer the dependency window , the better the quality of the word vectors , the author also found that this increases complexity and sometimes distant words are less related to the current word being modeled. The author used a window size of 10 in their original paper for training and results showed that the skip gram model outperformed CBOW in several experiments.

The original skip gram model trained on a corpus of 320 million words and 32k vocabulary size lasted for 8 weeks.

Source Text	Training Samples generated from source text
I will have orange juice and eggs for breakfast	(will, I) (will, have) (will, orange)
I will have orange juice and eggs for breakfast	(have, I) (have, will) (have, orange) (have, juice)
I will have orange juice and eggs for breakfast	(orange, will) (orange, have) (orange, juice) (orange, and)
I will have orange juice and eggs for breakfast	(juice, have) (juice, orange) (juice, and) (juice, eggs)
I will have orange juice and eggs for breakfast	(and, orange) (and, juice) (and, eggs) (and, for)
I will have orange juice and eggs for breakfast	(eggs, juice) (eggs, and) (eggs, for) (eggs, breakfast)
I will have orange juice and eggs for breakfast	(for, and) (for, eggs) (for, breakfast)

The word highlighted in yellow is the source word and the words highlighted in green are its neighboring words.

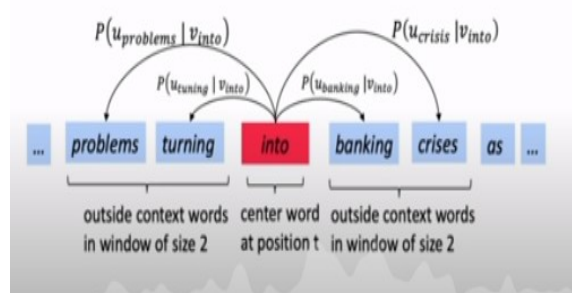
Given the sentence , “I will have orange juice and eggs for breakfast” and window size of 2, if the target word is juice , its neighboring words will be (have , orange and eggs). our input and target word pair would be (juice , have) , (juice , orange) , (juice , and) and (juice , eggs).

**Key Takeaways :-**

- 1) The continuous bag of words (CBOW) model uses context words to predict the target words , conversely the skip gram model uses the target words to predict the context words.
- 2) The word2vec algorithm causes words that have similar contexts to have similar word embedding (be close together) and words that have different contexts to be far apart.
- 3) Skip-gram is relatively slower than CBOW and usually works well with large corpus of training data
- 4) CBOW is faster to train than skip gram

**Question :-**

- 1) What is the representation of the input vectors before feeding it to the neural network , was it one hot vector or just a random vector ?
- 2) How does the vector representation of the word gets updated whenever our neural network learns , specially on the skip gram method?





With the context words in window of size 2, we want to know the meaning of the word “into” and so we are going to hope that its representation can be used in a way that will make precise to predict what word appear in the context of “into” because that is the meaning of “into” so we are going to try and make those predictions, see how well we can predict and then change the vector representation of words in a way that we can do the prediction better.

And once we just deal with “into”, we can go to the next word and we say okay, let’s take banking as a word, the meaning of banking is predicting the context in which banking occurs. so we will try and predict those words that occur around banking and see how we do and then move on again from there.

### WORD2VEC PREDICTION FUNCTION

In different words have components of the same sign, plus or minus, in the same positions, that dot product will be big and if they have different signs, one is big and one is small then the dot product will be a lot smaller (0 if they completely don’t have any similar meaning) so the dot product directly calculates similarity between words, where similarity is the sort of vectors looking the same. So we’re gonna have words that have similar vectors, close together in the vector space have similar meaning.

#### Word2vec: prediction function

Exponentiation makes anything positive

$$P(o|c) = \frac{\exp(u_o^T v_c)}{\sum_{w \in V} \exp(u_w^T v_c)}$$

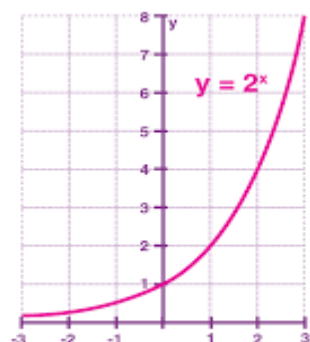
Dot product compares similarity of  $o$  and  $c$ .  
 $u^T v = u \cdot v = \sum_{i=1}^n u_i v_i$   
 Larger dot product = larger probability

Normalize over entire vocabulary to give probability distribution

- This is an example of the **softmax function**  $\mathbb{R}^n \rightarrow \mathbb{R}^n$ 

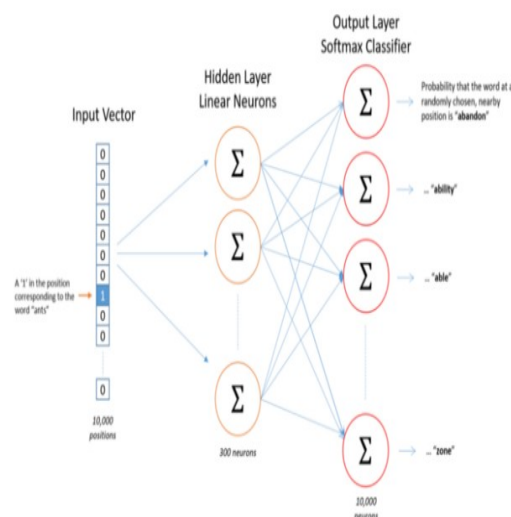
$$\text{softmax}(x_i) = \frac{\exp(x_i)}{\sum_{j=1}^n \exp(x_j)} = p_i$$
- The softmax function maps arbitrary values  $x_i$  to a probability distribution  $p_i$ 
  - “max” because amplifies probability of largest  $x_i$
  - “soft” because still assigns some probability to smaller  $x_i$

$\exp(u_o^T v_c)$  :- The exponential has this nice property no matter what number you stick in to it, because the dotproduct can be positive or negative, it’s gonna come out as a positive number. So if we want to get probability, that’s really good if we have positive numbers and not negative numbers, so that’s good.



$\sum \exp(u_o^T v_c)$  :- This is because we want to have probabilities and probabilities are meant to add up to 1, so we do that by sum up this quantity, for every different word in our vocabulary and we divide through by it so that normalizes it and turns in to a probability distribution.

The probability that is summed up to 1 is not with the all the context vectors surrounded that word but with the entire words in the vocabulary, the output of the skip gram network is a multi label classification problem, the output in the last node is the word with the highest probability in comparison with entire word from the vocabulary. So we will try to make predictions and using back propagation we will change the vector representation of the the center words in a way that makes our prediction better.



This is a Softmax function which maps arbitrary values  $x_i$  in to a probability distribution. The softmax is a mathematical function that converts a vector of numbers in to a vector of probabilities, where the probabilities of each value are proportional to the relative scale of each value in the vector. Softmax function is used to normalize the outputs, converting them from a weighted sum values in the probabilities that sum to one.

The name softmax refers to the softer version of hardmax, which is one hot encoded argmax, why are we calling the argmax as the hard version and softmax as the softer version?

Softmax is a mathematical function can be thought to be probabilistic or softer version of the argmax function. The term softmax is used because this activation function represents a smooth version of the winner takes all activation model in which the unit with the largest input has output 1 while all other have outputs 0.

From probabilistically perspective, if the  $\text{argmax}()$  function returns 1 and it returns 0 for the other two array indexes, giving full weight to index 1 and no weight to index 2 for the largest in the list  $[1, 3, 2] \Rightarrow [0, 1, 0]$ .

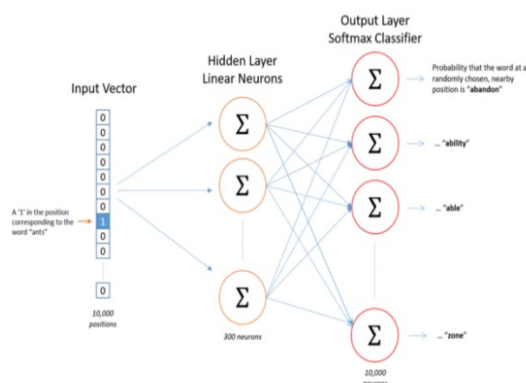
What if we are less sure and want to express the argmax probabilistically with likelihoods? This can be achieved by scaling the values in the list and converting them to probabilities, such that all values in the returned list sum to 1.0. This can be achieved by calculating the exponent of each value in the list and dividing by the sum of the exponent class.

Questions :-

- How do we represent the input representation of inputs before feeding them to the skip gram model?
- The cost function for the skip gram and CBOW architectures?
- Training skip gram and CBOW by optimizing the parameters
- How do we update our word representations in order to accurately predict the context vectors (in skip gram) and to get a good word embedding

### Details of Skip-Gram

The model is made of a single hidden layer and an output layer with a softmax classifier. The input of the model is a vector that has the size of the vocabulary, 0's everywhere except for a 1 at the position of the word we'd like to embed.



So when we multiply the first hidden layer weights with the input word, all of them will be zero except for the word that we want to embed. The output of the model is a Softmax layer that has the size of the vocabulary and for each word will be observed in the context of the input word. Softmax transforms to map the output as probabilities that sum to 1.

The hidden layer helps to choose the size of the vectors we'll later be using. If we set the hidden layer size to 500, the hidden layer will be a weight matrix of the size of the vocabulary, say 15,000 rows \* 500 columns. The hidden layer in this model is mainly operating as a look up table, that is selecting the row corresponding to the word vector. (This means if we have 500 node in our first hidden layer, that means we will extract 300d feature of that word.). Good for its word2vec, has used 300 dimensions for example.

By multiplying the feature vector of "dog" with an output layer of "walking", what we're computing here is the probability that if we pick a word randomly around "dog" this word is "walking" for example.

If the two words are different but happen in similar contexts, our model will learn similar word vectors for these words that is "engine" and "transmission".

All we need to do once the model has been trained is to drop the output layer. In deed we are only interested in the vector representation of the words and as in auto encoder, the second part of the network will not be used.

Key takeaways :-

- ✓ The input of the model is a vector that has the size of the vocabulary, 0's everywhere except for a 1 at the position of the word we'd like to embed.
- ✓ The first hidden layer helps to choose the size of the vector we'll later be using in the embedding.
- ✓ The weight matrix of the size of the vocabulary, say 15,000 rows \* 500 columns will be our featurized representation of the word. But since our 15,000 rows are one hot vector, only the row that we want to embed will have a dense representation, so the matrix will be changed to a row vector.
- ✓ At the beginning the word vector (the weight of the first hidden layer) literally going to start with a random vector for each word but then we literally going to change those vectors a little bit as we learn.
- ✓ Once the model has been trained we will drop the output layer, indeed we are only interested in the vector representation of the words and as in auto encoder the second part of the network will not be used.

- ✓ The only parameter of this model has is the vector space representation of the words.

- ✓ Glove Word Vector Algorithm

## NEGATIVE SAMPLING

---

Question :-

As we know each word has two vectors , pair of the center word (U) and the context word (V) , and the way we calculate the U and V vectors is we literally going to start with random vector for each word and then we literally going to change those vectors a little bit as we learn , so how do we update the vector representation of the context words as we did for the center words ?

### LIMITATION OF SKIP - GRAM

**Word2vec: prediction function**

Exponentiation makes anything positive

Dot product compares similarity of  $o$  and  $c$ .  
 $u^T v = u \cdot v = \sum_{i=1}^n u_i v_i$   
 Larger dot product = larger probability

$$P(o|c) = \frac{\exp(u_o^T v_c)}{\sum_{w \in V} \exp(u_w^T v_c)}$$

Normalize over entire vocabulary to give probability distribution

- This is an example of the **softmax function**  $\mathbb{R}^n \rightarrow \mathbb{R}^n$ 

$$\text{softmax}(x_i) = \frac{\exp(x_i)}{\sum_{j=1}^n \exp(x_j)} = p_i$$
- The softmax function maps arbitrary values  $x_i$  to a probability distribution  $p_i$ 
  - "max" because amplifies probability of largest  $x_i$
  - "soft" because still assigns some probability to smaller  $x_i$

The problem is computational speed. In particular for the softmax model every time we want to evaluate this probability , we need to carry out a sum of overall 10,000 words in our vocabulary and may be 10,000 isn't bad but if we're using a vocabulary of size 100 thousand or million , it gets really slow to sum up over this denominator every single time.

One solution for this is use Hierarchical Softmax classifier and what that means is , instead of trying to categorize something in to all 10,000 categories on one go , imagine if we have one classifier to tells you is that target word in the first 5000 words of the vocabulary or in the next 5000 words of vocabulary.

So we will eventually go down classifying until we're exactly what word is the leaf of this tree. Each of the interior nodes of the tree can be just a binary classifier and we don't need to sum over all the 10,000 words all of the vocab size in order to make a single classification.

In practice the Hierarchical Softmax classifier can be developed so that the common words tend to be on the Top , where as the less common words like durian can be put much deeper in the tree because we see the more common words more often and so we might need only a few traversals so get to the common words , where as less frequent words like durian much less often will put deeper in the tree.

Other Methods For Learning The Embedding

- ✓ Negative Sampling

# DEEP DIVE TO RECURRENT NEURAL NETWORKS

## NEURAL NETWORKS

We will learn a neural net fundamentals :-

- ➔ We concentrate on understanding deep , multi layer neural networks and how they can be trained (learned from data) using back propagation.
- ➔ we will look at an NLP classifier that adds context by taking in windows around a word and classifies the center word (not just representing it across all windows)

## OBJECTIVE :-

- word window classification , Neural Networks and calculus.
- Classification review
- Neural Networks Introduction
- Named Entity Recognition
- Binary True vs Corrupted Word Window Classification
- Matrix calculus introduction

## CLASSIFICATION SETUP AND NOTATIONS

So we assumed we have a training set where we have these vectors  $X$  of our  $x$  points and then for each one of them we have a class

So the input might be words or sentences, documents or something , they are  $d$ -dimensional vector , the  $y_i$  , the labels or classes that we want to classify to and we've got a set of  $c$  classes that we are trying to predict.

$X_i$  = are inputs , example :- words (indices or vectors ) , sentences , documents etc ...

$y_i$  :- are labels (one of the  $c$  classes ) , we try to predict for example

\* classes :- sentiment , named entities , buy/sell decision  
\* other words (language model)

\* Later :- Multi word sequences

## CLASSIFICATION INTUITION

The intuition is we got this vector space which we again have a 2d vector space which correspond to our  $X$  items and what we would want to do is we will look at the ones in our training sample and see which ones are green and the red ones as best as possible and that learned line is our classifier.

## RECURRENT NEURAL NETWORKS

- What does the name Recurrent refers to ?
- Why do we choose these model for sequential problems ?
- What are the problems with the standard neural networks or convolutional networks ?
- How does this neural networks patternize textual data ?

What does the name Recurrent refers to in Recurrent Neural Network ?

Recurrent ( Re-occurring , Re-appearing ) means that a new occurrence of something that happened or appeared before.

## WHY NOT STANDARD NEURAL NETWORKS

1) Input and outputs can be different length in different training examples , so there is no defined input and output shape. But one solution for this is to set a maximum length and work in according to that but this is still doesn't seem a good solution.

2) Doesn't share features learned across different positions of the text suppose we have an input word  $x_{<t>}$  these input text  $x_{<t>}$  doesn't learn features from the top or the bottom features.

Example :- 1) He said , "Teddy Roosevelt was a great president "

2) He said , " Teddy bears are on the sale "

In the first example , if  $X_{<3>}$  can't learn features from  $X_{<4>}$  it doesn't be sure that teddy is a name or not because in the second example we can see that Teddy is not a person's name.

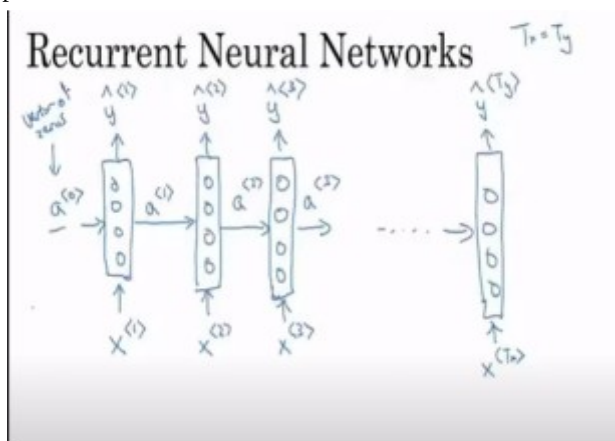
3) The number of parameters will be high (this is basically the problem of a standard neural network ) , previously we said that each of these is a 10,000 dimensional one hot vector , so this is just a very large input layer and if the total input size of the maximum number of words times 10,000

## HOW DOES RNN SOLVES THIS PROBLEM ?

So recurrent neural networks solve all of these problems. How recurrent neural networks solve these problems and how it suits for a sequential data.

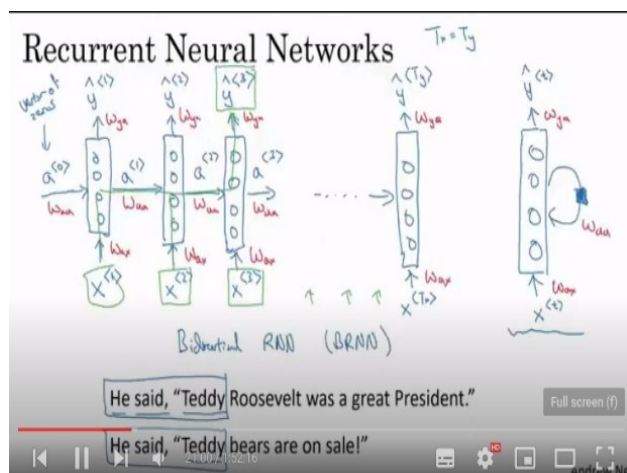


**Recurrent Neural Networks :-** so if we are reading a sentence from left to right , the first word you read say  $X<1>$  and take this first word and feed it to the neural network layer and make a prediction that it's part of a person's name or not.

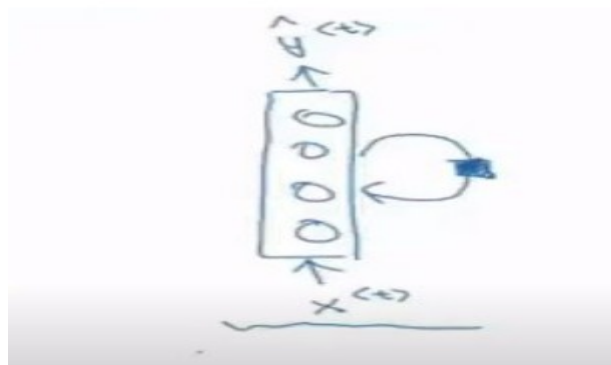


What a neural network does is when then goes on to read the second word in the sentence  $x<2>$  , instead of predicting  $y<2>$  using  $x<2>$  , it also gets to input some information from the earlier network. **But how does this works , are we gonna add the previous network with the next network , what is the math behind this ?** In our example  $T_x = T_y$  , the length sequence is equal to the output sequence , this is no always happened so that the Architecture will also change.

And to kick of the whole thing we'll also have some made up activation with a time step zero , this is usually a vector of zeros and some researchers initialize  $a<0>$  randomly.



On some books or research papers we will find a neural network architecture like this



It's difficult to interpret this but it's the same as the one in the above (we just need to unroll it ).

The parameters which governs the connection from  $X<t>$  to the hidden layer is gonna be denoted as  $W_{ax}$ . we are using the same parameter for all of the time steps. What advantage will this provide and how is that learn different features across different time steps ?

The horizontal connections will be governed by some set of parameters  $W_{aa}$  (the same parameters for all of the time steps ). The output predictions will be governed by some set of parameters called  $W_{ya}$  (same parameters for all of the time steps ). **Why are we using the same parameters in each of the time steps ?**

Note in the recurrent neural network , when making a prediction for  $Y<3>$  , it gets information not only from the  $X<3>$  but also from  $X<2>$  and  $X<1>$  because the information from  $X<1>$  pass through on the way to help the prediction with  $y<3>$ .

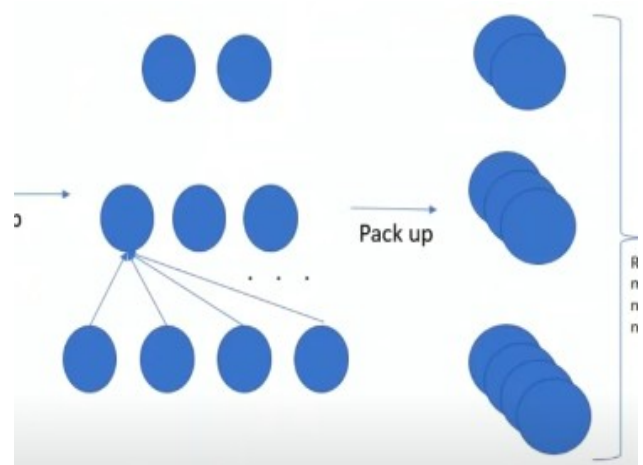
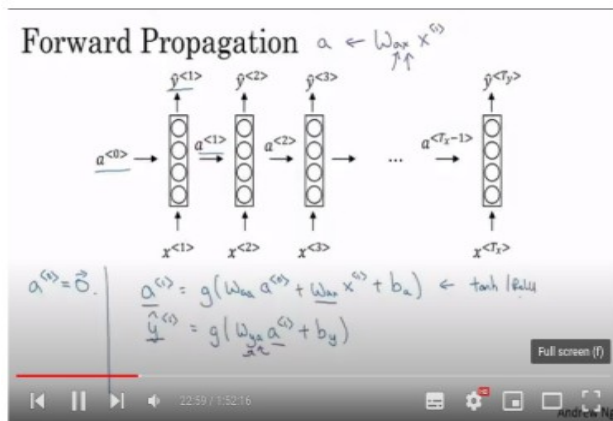
Now the weakness of this RNN is that , it only uses the information that is earlier in the sequence to make prediction , in particular when predicting  $y<3>$  , it doesn't use information from the words  $x<4>$  ,  $x<5>$  ,  $x<6>$  and so on.

Example :- He said , "Teddy Roosevelt was a great president "

In order to decide whether not the word teddy is a part of a person's name , it'd be really useful to know just information from the first two words ( He said ) but to know information from the later words in the sentence as well.

Example :- He said , "Teddy bears are on sale " - sentences like these could also happen , that given just the first three words , is not possible to know for sure whether the word teddy is a part of a person's name or not. In the first example it is but on the second example it's not but you can tell the difference if you look only at the first three words. We will address these problems using Bi-Directional Recurrent Neural Networks (BRNN) but this is a uni-directional neural network architecture.

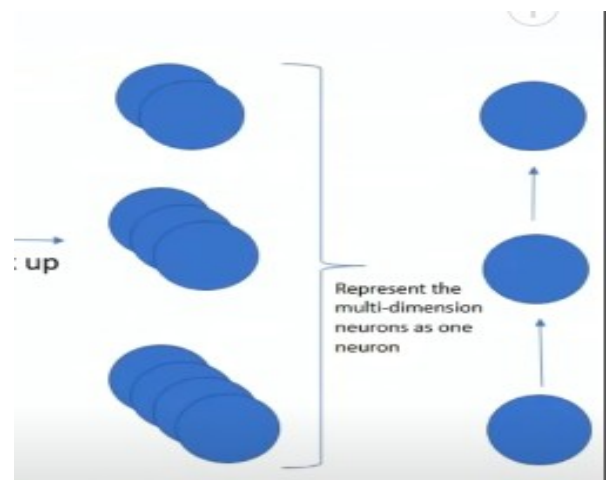
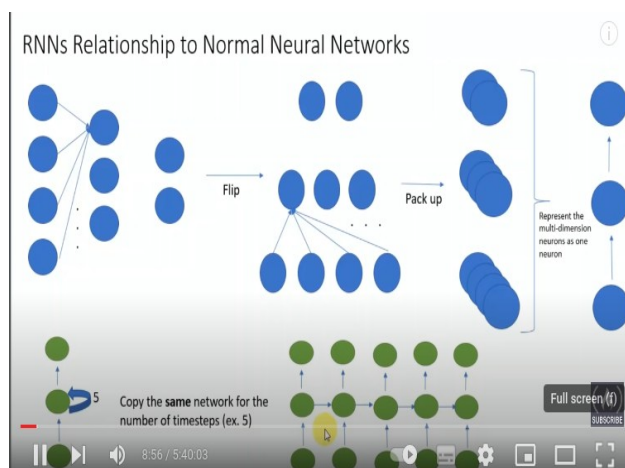
## Forward Propagation In Time Of RNN



Represent the packed up multi-dimensional neurons as one neuron.

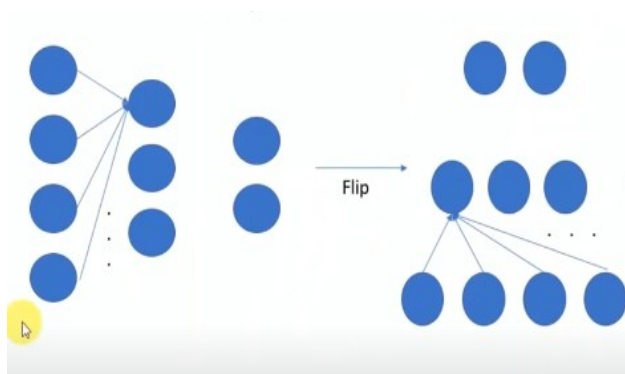
What is the reason for the name “Forward Propagation In Time ? ” because our RNN will act at each and every time steps like at  $t = 1$  will pre process one word and at  $t = 2$  will pre process another word.

How does the architecture changed from the standard Neural Network to recurrent Neural Network

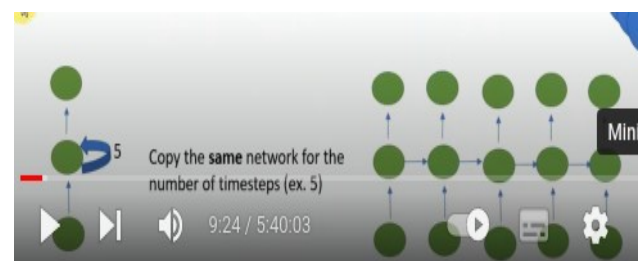


So copy of the same network for the number of time steps , where each time accepts the previous hidden state and the current time step.

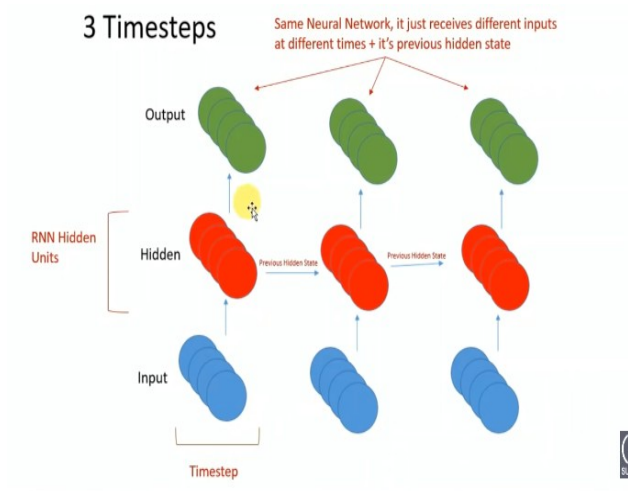
First we are going to flip the standard neural network



Packed up the flipped neural networks



This is just a multiple copy of the standard neural network.

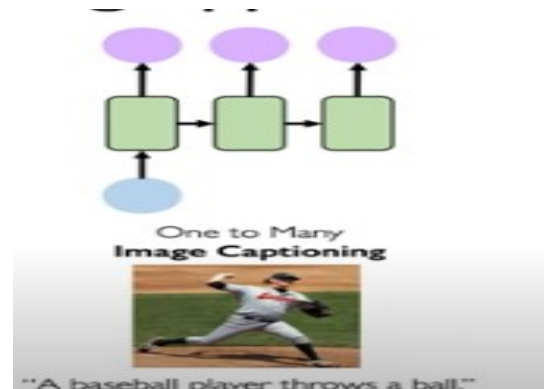
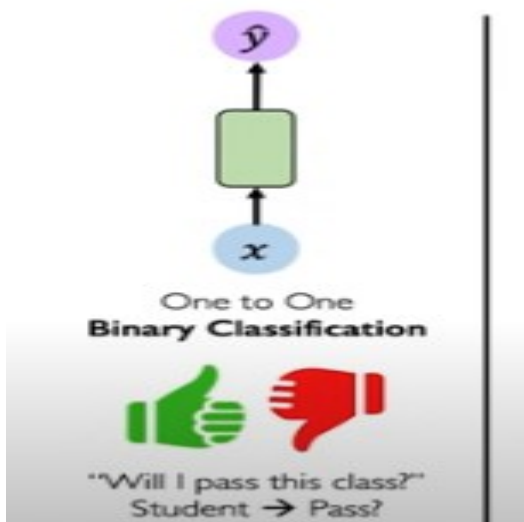


well so for example let's consider the case where there's a sentence as input to our model and that defines a sequence where the words in the sentence are the individual time steps in that sequence and at the end our task is to predict one output which is going to be the sentiment or feeling associated with that sequence input and you can think of this problem as having a sequence input , single output or as many to one sequence problem.

We can also consider the converse case where now our input does not have that time dimension so for example we're considering a static image and our task is now to produce a sequence of output for example a sentence caption that describes the content in this image.

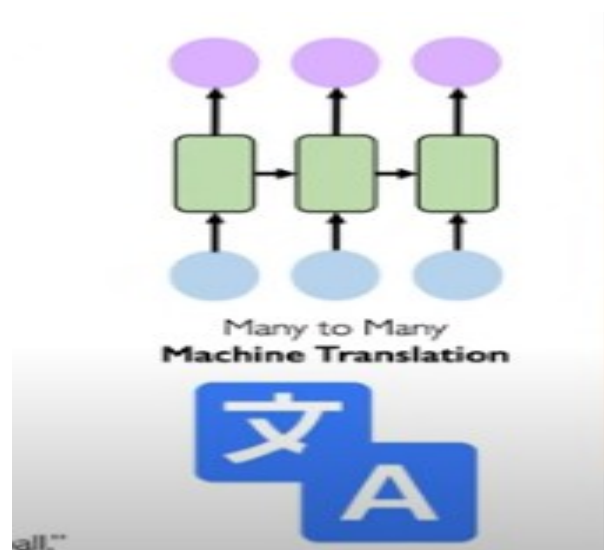
## DEEP DIVE TO RNN FROM MIT

Feed Forward neural networks operate in this one to one manner going from a fixed and static input to a fixed and static output , for example in the use case of binary classification we were trying to build a model that given a single input of a student in the class could be trained to predict whether or not that student was going to pass or not and in this type of example there's no time component , there's no inherent notion of sequence or sequential data.



And we can think of this as one to many sequence modeling problem. Finally we can also consider this situation on many to many where we're now translating from a sequence to another sequence and perhaps one of the most well known example of this is machine translation where the goal is to train a model to translate sentences from one language to another.

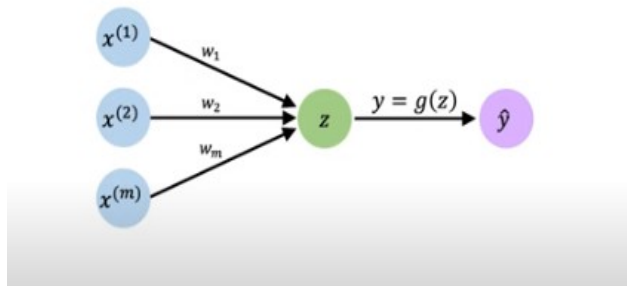
When we consider sequence modeling we now expand the range of possibilities to situations that can involve temporal inputs and also potentially sequential outputs as



## NEURONS WITH RECURRENCE

So what changes need to be made to our neural network architecture in order to be able to handle sequential data.

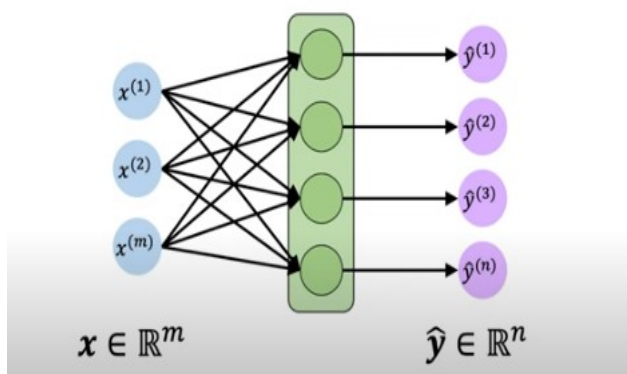
### The Perceptron Revisited



We define perceptrons as a set of inputs which we can call  $x_1$  through  $x_m$  and each of these numbers are going to be multiplied by a weight matrix and then they are going to be added together to form this internal state of the perceptron which we'll say  $z$  and then this value  $z$  is passed through a non-linear activation function to produce a predictive output  $\hat{y}$  and remember in perceptrons there are multiple inputs as being from a single time step in our sequence.

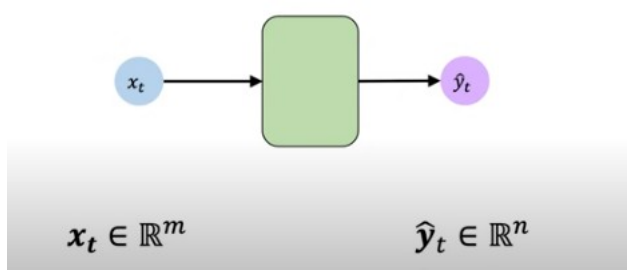
We saw how we would extend from the single perceptron to now a layer of perceptrons to yield multi-dimensional outputs.

### Feed-Forward Networks Revisited



For example here we have a single layer perceptron in green, taking three inputs in blue and predicting four outputs shown in purple but once again this does not have

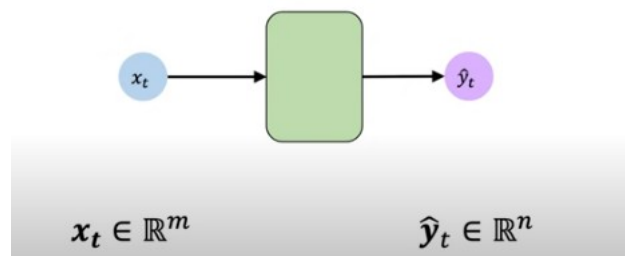
### Feed-Forward Networks Revisited



a notion of time or sequence because our inputs and outputs are a fixed time step in our sequence.

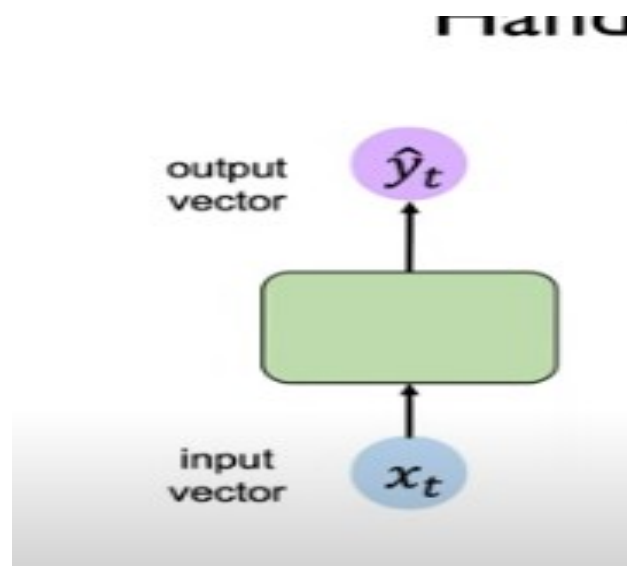
So let's simplify the diagram.

### Feed-Forward Networks Revisited



We'll collapse that hidden layer down to this green box and our input and output vectors will be as depicted here and again our inputs  $x$  are going to be some vectors of length  $m$  and our outputs are going to be length  $n$  but still we're considering the input as just a specific time denoted by  $t$ .

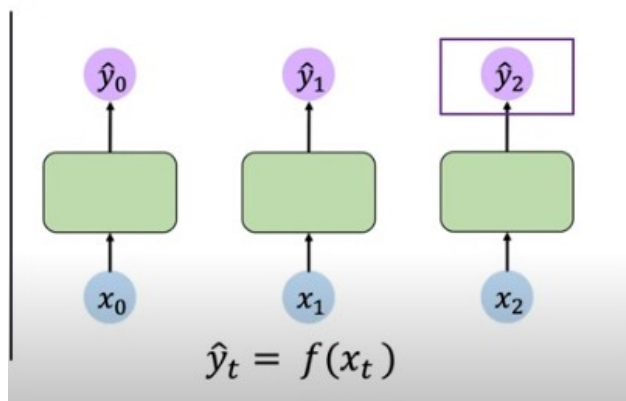
And even with this simplified representation of a feed-forward network we could naively try to feed a sequence in to this model by just applying that same model over and over again for each time step in our sequence, to get a sense of this and how we could handle these individual inputs across different time steps but let's first rotate the above diagram.



So now again we have an input vector  $x_{<t>}$  for some time step  $<t>$ , we feed it in to our neural network and get an output vector at that time step but since we're interested in sequential data let's assume we don't have a single time step and we have multiple individual time steps which start from let's say time zero, the first time step in our sequence and we could take that input at that



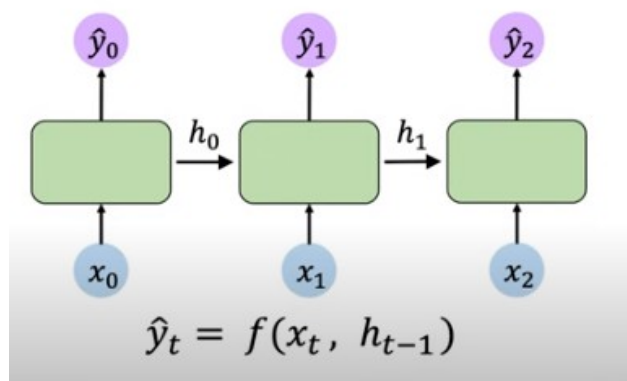
time step , treat it as this isolated point in time , pass it to the model and generate a predictive output and we could do that for the next time step again treating it as something isolated and same for the next.



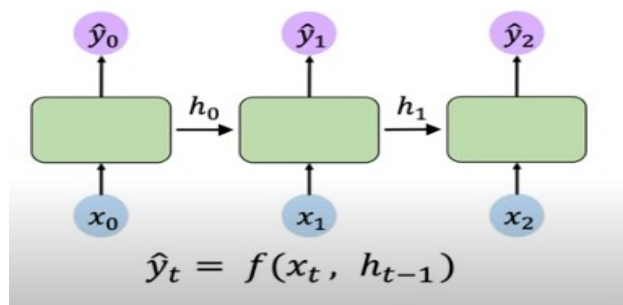
Here all of the models depicted here are replicas to each other with different inputs at different time steps but we know that our output vector  $\hat{y}_t$  at a particular time step  $t$  is just going to be a function of the input at that time step but let's take a step back here for a minute , if we're considering sequential data it's probably very likely that the output or the label at a later time step is going to somehow depend on the inputs at prior time steps so what we're missing here is by treating these individual time steps as individual isolated time steps.

This relationship that's inherent to sequence data between inputs earlier on in the sequence to what we predict later in the sequence , so how could we address this , what we really need is a way to relate the computations and the operations that the network is doing at a particular time step to both the prior history of its computation from prior time steps as well as the input at that time step.

What we'll consider is linking the information and the computation of the network at different time steps to each other , specifically we are going to introduce this internal memory or cell state which we denote here as  $h_{t-1}$  and this is going to be passed on time step to time step across time and the key idea here is by having a



recurrence relation we're capturing some notion of memory of what the sequence looks like

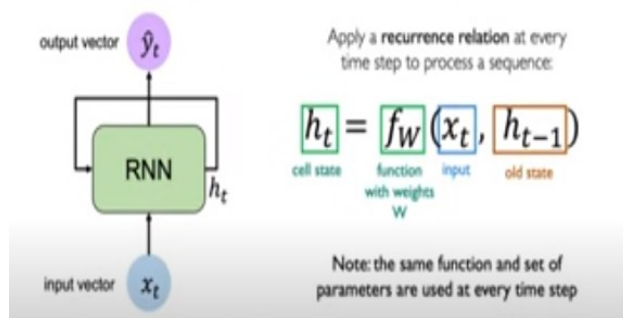


What this means is now the network's output predictions and it's computations are not only a function of the input at a particular time step but also the past memory of cell state denoted by  $h$  , that is to say that our output depends on both our current inputs as well as the occurred and we can define this relationship via these functions that map inputs to outputs and these functions are standard neural network operations that we know.

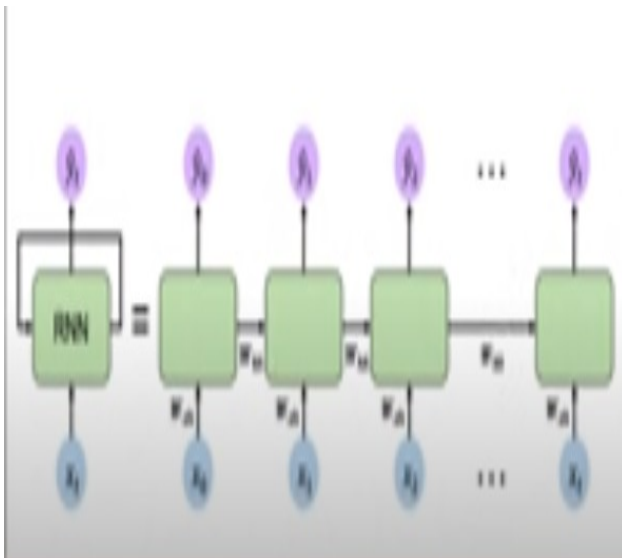
So once again our output, our prediction is going to depend not only on the current input at a particular time step but also on the past memory.

$$\hat{Y}^t = f(X_t, h_{t-1})$$

And because as we see in this relation here our output is now a function of both the current input and the past memory at a previous time step , this means we can describe this neurons via a recurrence relation which means that the we have the cell state that depends on the current input and again on the prior cell states.



And the depiction on the right on this line shows these individual time steps being sort of unrolled across time but we could also depict the same relationship by this cycle and this is shown on the loop on the left of the slide , which shows on the loop on the left of the slide , which shows this concept of a recurrence relation.



## FORMALIZING RECURRENT NEURAL NETWORKS

The key idea is that this RNN's maintain this internal state  $h_{<t>}$  which is updated at each time step as the sequence is processed and this is done by this recurrence relation which specifically defined how the state is updated at the time step.

Specifically we define this internal cell state  $h_{<t>}$  and that internal state is going to be defined by a function that can be parameterized by a set of weights  $w$ , which are actually trying to learn over the course of training such a network and that function  $f$  of  $w$  is going to take as input both the input at the current time step  $x$  of  $t$  as well as the prior state  $h_{<t-1>}$

$$h_{<t>} = f_w(x_t, h_{t-1})$$

How do we actually find and define this function ? again it's going to be parameterized by a set of weights that are going to be specifically what's learned over the course of training the model and a key feature of RNN's is that use this very same function and this very same set of parameters at every time step of processing the sequence and of course the weights are going to change over time over course of training and later on we'll see exactly how but at each iteration of training that same set of weights is going to be applied to each of the individual time steps in the sequence.

The RNN computation includes both this internal cell state update to  $h_{<t>}$  as well as the output prediction itself.

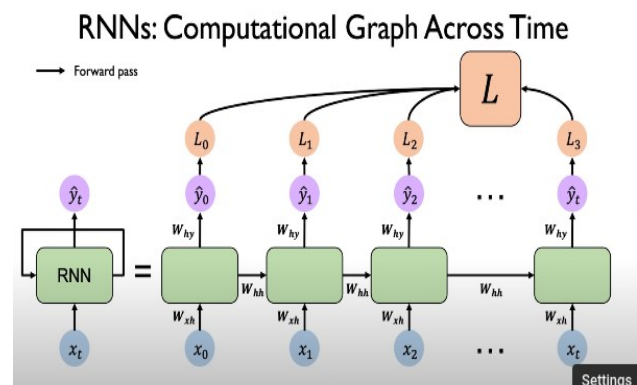
So we are going to consider our input vector  $X_{<t>}$  and we're next going to apply a function to update the hidden state and this function is a standard neural network operation and again because this internal cell state  $h_{<t>}$  is going to depend on both the input  $x_{<t>}$  as well as the prior cell state  $h_{<t-1>}$ , we are going to add the result and then apply a non linear activation function which in case is going to be a hyperbolic tangent ( $\tanh$ ) to the sum of these two terms to actually update the value of the hidden state.

$$h_t = \tanh(W_{hh}^T h_{<t-1>} + W_{xh}^T X_t)$$

and then to generate our output at a given time step, we take that interval hidden state, multiply it by a separate weight matrix which inherently produces a modified version of this internal state and this actually forms our output prediction so this give us the mathematics behind how the RNN can actually update its hidden state and also produce a predictive output.

$$\text{Output Vector} = y^t = W_{hy}^T h_t$$

So far we've seen RNN's being depicted as having these internal loops that feed back on themselves and we've also seen how we can represent this loop as being unrolled across time where we can start from a first time step and continue to unroll the network across time up until time step  $t$  and with in this diagram we can also make explicit the weight matrices starting from the weight matrix that defines how the inputs at each time computation as well as the weight matrices that define the relationship between the prior hidden state and the current hidden state and finally the weight matrix that transforms the hidden state to the output at a particular time step.



And again to re-emphasize in all of these cases for all of these weight matrices, we are going to be reusing the same weight matrices at every time step in our sequence (because we are re-using the same neural network again and again) **but what about the features? did every word extract the same set of feature? obviously not? so how?**

Now when we make a forward pass through the network we are going to generate outputs at each of those individual outputs we can derive a value for the loss and then we can sum all of these losses from the individual time steps together to determine the total loss which will be ultimately what is used to train our RNN.

Again with feed forward or traditional neural networks we're operating in this one to one manner going from a static input to a static output, in contrast with sequences we can go from a sequential input where we have many time steps defined sequentially over time feed them in to a recurrent neural network and generate a single output like a classification of sentiment or emotion associated with a sentence.

## Sequence Modeling : Design Criteria

To model sequences we need to ,

- 1) We need to be able to ensure that our recurrent neural network or any machine learning model that we may be interested in will be equipped to handle variable length sequences because not all sentences , not all sequences are going to have the same length so we need to have the ability to handle this variability.
- 2) We also need to have this critical property of being able to track long term dependencies in the data and to have a notion of memory and associated with aht
- 3) Have this sense of order and have a sense of how things that occur previously or earlier on in the sequence affect what's going to happen or occur later on and to do this we can achieve both points two and three by using weight sharing and actually sharing the values of the way matrices across the entire sequence.

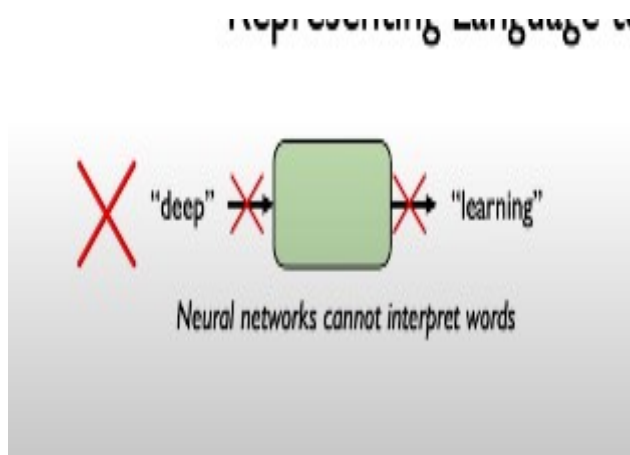
Recurrent Neural Networks do indeed meet all these sequence modeling design criteria.

### WORD PREDICTION PROBLEM

Let's see a sequence modeling problem , given some series of words in a sentence our task is going to be to predict the most likely next word to occur in that sentence.

So let's suppose we have this sentence as an example "This morning I took my cat for a " **walk** " and our task is let's say we're given these words " This morning i took my cat for a " and we want to predict the next word in the sentence "walk" and our goal is to try build a recurring neural network to do exactly this.

What's our first step to tackle this problem ? the first consideration before we even get started with training our model is how we can actually represent language to a neural network so , let's suppose we have a model where we input the word "deep" and we want to use the neural network to predict the next word "learning".



## Handle Variable Sequence Lengths

Let's predict this task of trying to predict the next word in a sentence , we can have very short sentences where driving the meaning of our prediction are going to be very close to each other like for example " The food was great " but we could also have a longer sequence like " we visited a restaurant for lunch " or even longer sequence like "we were hungry but cleaned the house before eating " , where the information that's needed to predict the next word occurs much earlier on and the key requirement for our recurrent neural network model is the ability to handle these inputs of varying length.

Feed forward networks are not able to do this because they have inputs of fixed dimensionality and then those fixed dimensionality inputs are passed in to the next layer.

In contrast RNN are able to handle variable sequence lengths and that is because those differences in sequence lengths are just differences in the number of time steps that are going to be input and processed by the RNN , so RNN's meet this first design criteria.

### MODEL LONG TERM DEPENDENCIES

Recurrent Neural Networks are able to achieve because they have this way of updating their internal cell state via the recurrence relation we previously discussed which fundamentally incorporates information from the past state in to the cell state update.

### Capture Differences in Sequence Order

Next we need to be able to capture differences in sequence order which could result in differences in the overall meaning or property of a sequence. For example in this case where we have sentences that have opposite semantic meaning but have the same words with the same counts just in a different order.

\* The food was good , not bad at all

Vs

\* The food was bad , not good at all

And once again the cell state maintained by an RNN depends on its past history which helps us capture these sorts of differences because we are maintaining information about past history and also reusing the same weight matrices across each of the individual time steps in our sequence.

### Back Propagation Through Time

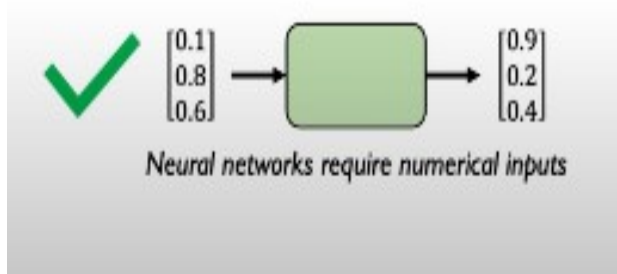
Let's recall how we can actually train feed forward models using the back propagation algorithm.

We first take a set of inputs and make a forward pass through the network going from input to output and to train the model we back propagate gradients back through the network and we take the derivative of the loss with

respect to each weight parameter in our network and then adjust the parameters, the weights in our model to minimize that loss.

What could be the issue here in terms of how we are passing these in this input to our network, remember that neural networks are functional operators, they execute mathematical operations on their inputs and generate numerical outputs as a result so they can't really interpret and operate on words if they're just passed in as words, so what we have here is just simply not going to work.

Instead Neural Networks require numerical inputs that can be a vector or an array of numbers such that the model can operate on them to generate a vector or array of numbers as the output



Now we know that we need to have a way to transform language in to this vector or array based representation, how exactly are we going to go about this?

The solution we're going to consider is this concept of embedding which is this idea of transforming a set of identifiers for objects effectively indices in to a vector of fixed size that captures the context of the input.

Let's again turn back to that example sentence that we've been considering, "I took my cat for a walk", we want to be able to map any word that appears or could appear in our body of language to a fixed size vector so our first step is going to be generate a vocabulary which is going to consist of all unique words in our set of language, we can then index these individual words by mapping individual unique words to unique indices and these indices can then be mapped to a vector embedding, one way we could do this is by generating sparse and binary vectors that are going to have a length that's equal to the number of unique words in our vocabulary such that we can then indicate the nature of a particular word by encoding this in the corresponding index and encoding language data and it's called a one hot encoding.

Another way we could build up these embeddings is by actually learning them so the idea here is to take our index mapping and feed that index mapping in to a model like a neural network model such that we can transform such index mapping across all the words of our vocabulary to a vector of a lower dimensional space where the values of that vector are learned such that words that are similar to each other have similar embeddings.

These two distinct ways in which we can encode language data and transform language data in to a vector representation that's going to be suitable for input to a neural network.

### Sequence Modeling Design Criteria

To model sequences, we need to

- 1) Handle variable length sequences
- 2) Track long term dependencies
- 3) Track long term dependencies
- 4) Share parameters across the sequence

So hopefully going through this example of predicting the next word in a sentence with a very particularly common type of sequential data being language data, this shows how sequential data being language data, this shows how sequential data more broadly can be represented and encoded for input to RNNs and how RNNs can achieve these set of sequence modeling design criteria.

For RNN's as we talked through earlier are forward pass through the network consists of going forward across time and updating the cell state based on the input as well as the previous state generating an output and fundamentally computing the loss values at the individual time steps in our sequence and finally summing those individual losses to get the total loss.

Instead of back propagating errors through a single feed forward network at a single time step in RNNs those errors are going to be back propagated from the over all loss through each individual time step and then across the time step all the way from where we are currently in the sequence to the beginning and this is the reason why it is called back propagation through time because as you can see all the errors are going to be flowing back in time for the most recent time step to the very beginning of the sequence.

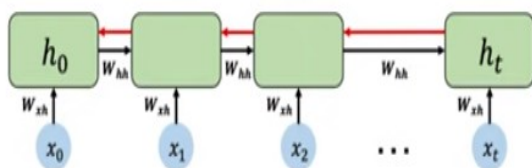
### Standard RNN Gradient Flow

Now if we expand this out and take a closer look at how gradients can actually flow across this chain of repeating recurrent neural network module, we can see that between each time step we have to perform matrix multiplication that involves the weight matrix  $W_{hh}$  and so computing the gradient with respect to the initial cell state  $h_{<0>}$  is going to involve many factors of this weight matrix and also repeated computation of the gradients with respect to this weight matrix.

This can be problematic for a couple of reasons, the first being that if we have many values in this series this chain of matrix multiplications where the gradient values are less or greater than 1 or the weight values are greater than 1, we can run in to a problem that's called the exploding gradient problem where our gradients are going to become extremely large and we can't really optimize and the solution here is to do what is called gradient clipping effectively.



## Standard RNN Gradient Flow



We can also have the opposite problem where now our weight values or our gradients are very very small and this can lead to what is called the vanishing gradient problem.

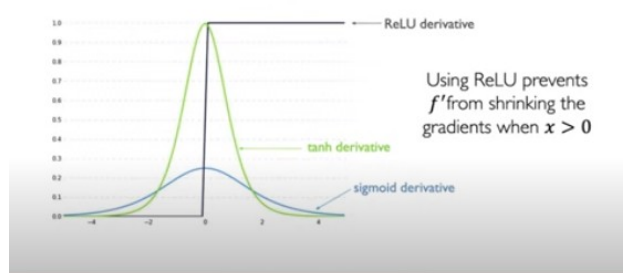
When gradients become increasingly smaller and smaller such that we can no longer effectively train the network and there are three ways to address vanishing gradient problem. First by cleverly choosing our activation function or by smartly initializing our weight matrices and finally we've discuss how we can make some changes to the network architecture itself , to alleviate this vanishing gradient problem (LSTM or GRU)

### The problem of Long Term Dependencies

Why are vanishing gradient problem ? Let's imagine we keep multiplying a small number something in between 0 and 1 by another small number over time , that number is going to keep shrinking and shrinking and eventually it's going to vanish and what this means when this occurs for gradients is that , it's going to be harder and harder to propagate errors from our loss function back in to the distant past because we have this problem of the gradients becoming smaller and smaller and ultimately lead to is we're going to end up biasing the weights and the parameters of our network to capture shorter term dependencies in the data rather than long term dependencies.

How can we get around this , the first trick we're going to consider is pretty simple we can smartly select the activation function our networks use , specifically what is common done is to use a ReLU activation function where the derivative of this activation function is  $> 1$  for all instance in which  $x$  is greater than zero and this helps the value of the gradient with respect to our loss function to actually shrink.

## Trick #1: Activation Functions



Another thing we can do is to be smart in how we actually initialize the parameters in our Network and we can specifically initialize the weights to the identity matrix to be able to try to prevent them from shrinking to zero completely and very rapidly during back propagation

## Trick #2: Parameter Initialization

Initialize **weights** to identity matrix

$$I_n = \begin{pmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 1 \end{pmatrix}$$

Initialize **biases** to zero

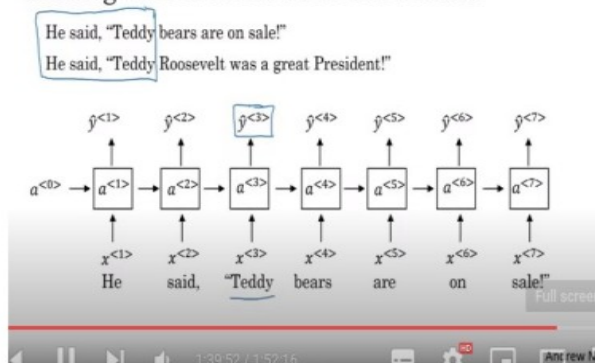
This helps prevent the weights from shrinking to zero.

Our final solution is to use a sort of more complex recurrent unit that can more effectively track long term dependencies in the data by intuitively you can think of it as controlling what information is passed through and what information is used to update the actual cell.

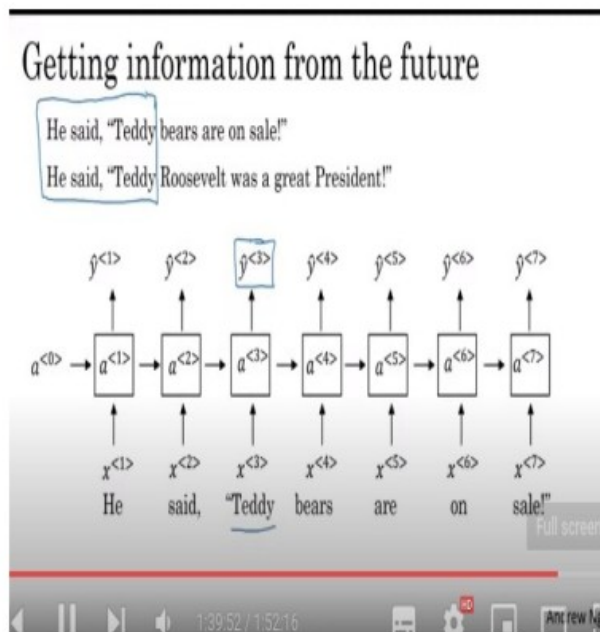
### Bi Directional Recurrent Neural Networks

Getting information from the future :-  
- He said , "Teddy bears are on sale !"

## Getting information from the future



- He said , “Teddy Roosevelt was a great president ”



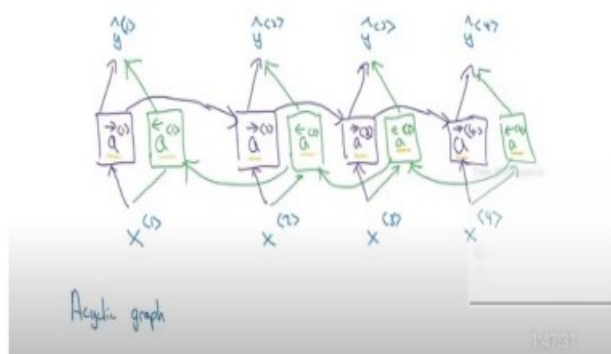
But one of the problems of this network is that to figure out whether the third word teddy is a part of a person’s name is not enough to look at the first part of the sentence , so to tell  $y^{<3>}$  is 0 or 1 , we need more information that just the first three words because the first three words doesn’t tell if they’re talking about teddy bears or the former president Teddy Roosevelt.

So this is a uni-directional or a forward directional only RNN and in our above example , we just made a standard RNN but it can be GRU or LSTM.

**Bi Directional RNN** :- These networks hidden layer will have a forward recurrent component and a backward recurrent component.

These networks has a forward component , so we will put a right arrow on their top of the activations to denote that it is a forward component.

## Bidirectional RNN (BRNN)



So the prediction  $y^{<t>}$  will be calculated as :-

$$y^{<t>} = g(w_y[a^{<t>}(\text{forward}), a^{<t>}(\text{backward})], b_y)$$

So if we look at the prediction at time step 3 then information from  $X^{<1>}$  can flow through  $a^{<3>}$  then  $y^{<3>}$  , so information from  $x^{<1>}$  ,  $x^{<2>}$  and  $x^{<3>}$  they are all taken to account where as information from  $x^{<4>}$  flow through a backward  $a^{<4>}$  to  $a^{<3>}$  , then to  $y^{<3>}$  , so the prediction at time 3 , to take the input both information from the past as well as information from the present ( $X^{<3>}$ ) which both the forward and backward goes to the same step.

Notice that this network defines a cyclic graph then given an input sequence  $X^{<1>}$  to  $X^{<4>}$  then go to the forward to compute  $a^{<1>}$  ,  $a^{<2>}$  ,  $a^{<3>}$  and backward sequence will start to computing a backward and then go back and compute  $a^{<4>}$  ,  $a^{<3>}$  and notice that we are computing network activations , this is not back propagation , this is forward propagation but bi-directional , it goes left to right and part of the computation goes from the right to the left.

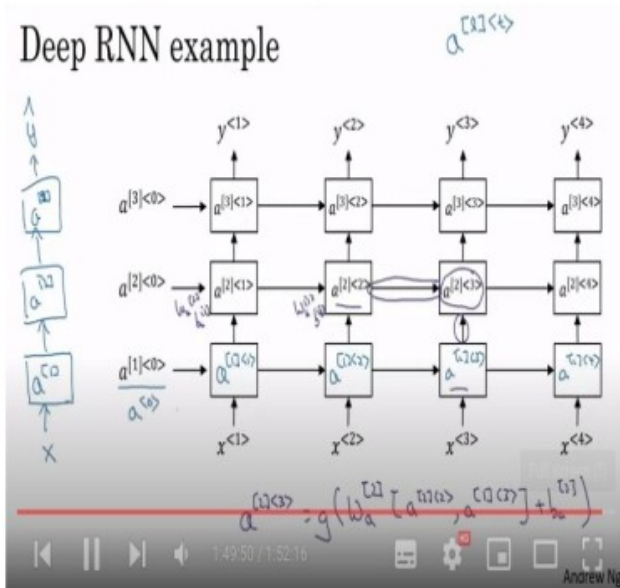
For this example we just used a standard RNN but they can also be GRU and LSTM.

The disadvantage of Bi-Directional RNN is that , you do need the entire sequence of data before you can make a prediction , so for example , if we are building a speech recognition system , so BRNN will let you take in to account the entire speech utterance , so if we use BRNN we need to wait for the person to stop talking to get the entire utterance before we actually process it and make a speech recognition prediction.

## Multi Layer Recurrent Neural Networks (Deep RNN)

For learning very complex function it’s sometimes useful to stack multiple layer’s of RNN together to build even deeper versions of these models.

RNN but generally Deep RNN's are computationally expensive.



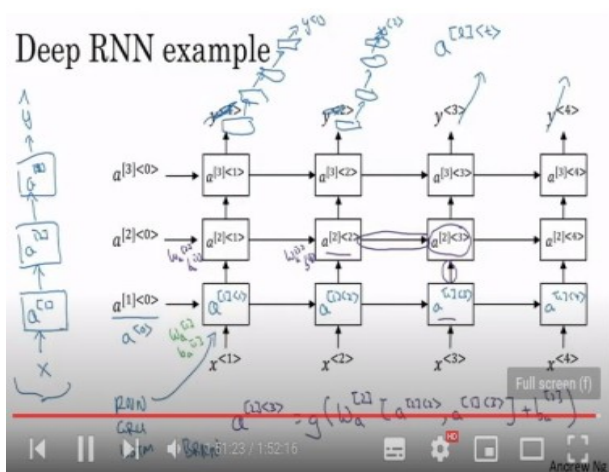
we changed a little bit of notation , instead of only saying  $a^{<0>}$  we just added a square bracket notation to denote the layer  $a^{[l]<t>}$  , which means an activation with a layer  $l$  and a time step  $t$ .

So let's look an example :- to see how these value is computed , let's select  $a^{[2]<3>}$  , has two inputs from  $a^{[2]<2>}$  (input from the left) and an output coming from the bottom ( $a^{[1]<3>}$ ) so to compute an activation function  $G$  , applied to a weight matrix.

$$a^{[2]<3>} = g(w_a^{[2]} [a^{[2]<2>}, a^{[1]<3>}] + b_a^{[2]})$$

For RNN's having three layer is already quite a lot because of the temporal dimension these networks can already get is quite big , even if you have just a small handful of layers and we don't usually see these stacked up (people's don't use it that much).

What we see that mostly people are using these one



A deep network but they are not connected horizontally  
In these example we just saw a standard RNN but we can also use GRU , LSTM or we can build a deep version of

## Natural Language Processing , Computer Vision and Unsupervised Learning With Transformers

You probably seen some astonishing demos of these language models such as GPT-3 , which given a short prompt such as “a frog meets a crocodile” can write a whole story. Although it’s not quite Shakespeare yet , it’s sometimes hard to believe that these were written by an artificial neural network. The revolution goes far beyond text generation (language models) , it encompasses the whole realm of natural language processing (NLP) , from text classification (Semantic Analysis) to summarization , machine translation , question answering , chat bots , natural language understanding (NLU) and more. Wherever there’s language , speech or text , there’s an application for NLP. You can already ask your phone for tomorrow’s weather or chat with a virtual desk assistant to troubleshoot a problem or get meaningful results from search engines that seem to truly understand your query. But the technology is so new that the best is probably yet to come.

**Question :- What is the difference between question answering and chat bots ?**

Chatbots and Q&A systems differ in their complexity as well as use cases. Chatbots can answer various questions asked during an interactive conversation. Interactive conversation means the system keeps a track of questions asked earlier and can engage in longer conversations. They have a sort of memory which help answer in a more friendlier manner. Also , they retrieve information such as weather , stock prices from various resources. Hence their ability is far beyond Q&A systems in this sense. On the other hand Q&A systems are programmed to answer questions only from a particular source of information , sometimes questions belonging to a common topic. The “Ask Question” dialog found on most websites is an example of such a system. They could be thought of a search engine which only works for a specific topic.

Like most advances in science , this recent revolution in NLP rests upon the hard work of

hundreds of unsung heroes. But three key ingredients of its success do stand out :-

- ✓ The transformer is a neural network architecture proposed in 2017 in a groundbreaking paper called “Attention Is All You Need ” , published by a team of Google researchers. In just a few years it crushed previous architectures that were typically based on recurrent neural networks (RNNs). The Transformer architecture is excellent at capturing patterns in long sequences of data and dealing with huge datasets -- so much so that its use is now extending well beyond NLP , for example to image processing tasks.
- ✓ In most projects , you won’t have access to a huge dataset to train a model from a scratch. Luckily , it’s often possible to download a model that was Pre-Trained on a generic dataset. All you need to do is fine tune it on your own (much smaller) dataset. Pretraining has been a main stream in image processing since the early 2010s but in NLP it was restricted to contextless word embedding (that is a dense vector representation of individual words ) For Example :- the word “bear” had the same pretrained embedding in “teddy bear” and in “ to bear ”. Then in 2018 , several papers proposed full blown language models (like GPT-3 , BERT) that could be pre-trained and fine tuned for a variety of NLP tasks , this completely changed the game.
- ✓ Model hubs like HuggingFace’s have also been a game changer. In the early days , pretrained models were just posted anywhere , so it wasn’t easy to find what you needed. Murphy’s law guaranteed that PyTorch users would only find TensorFlow models and vice versa. And when you find a model , figuring out how to fine-tune it wasn’t easy. This is where Hugging Face’s Transformers library comes in , it’s open source , it supports both TensorFlow and PyTorch and it makes it easy to download a state of the art pretrained model from the Hugging Face Hub , configure it for your task , fine tune it on your dataset and evaluate it. The library and its ecosystem are expanding beyond NLP , image



processing models are available too. You can also download numerous datasets from the Hub to train or evaluate your models.

#### Question :-

- ✓ What is pretrained embedding ? what was the limitation ?
- ✓ What are Pretrained Word Embeddings ?
- ✓ Why do we need pre trained word embeddings ?
- ✓ What are the Different Pretrained word embeddings ?
  - ✓ 1) Google's Word2Vec
  - ✓ 2) Stanford's Glove Word Vector Algorithm
- ✓ What is Hugging Face ?
- ✓ What is Hugging Face Hub ?
- ✓ What is Murphy's Law

Pretrained Word Embeddings are the embeddings learned in one task that are used for solving another similar task. These embeddings are trained on large datasets , saved and then used for solving other tasks. That's why pretrained word embeddings are a form of transfer learning.

Transfer learning as the name suggests is about transferring the learnings of one task to another. Learnings could be either weights or embeddings . In our case learnings are the embeddings. Hence , this concept is known as pretrained word embeddings (Our updated parameters will be the vector representations of the words ). In the case of weights the concept is known as a pretrained model.

Why do we need pretrained Word Embeddings ? Pretrained word embeddings capture the semantic and syntactic meaning of a word as they are trained on a large datasets. They are capable of boosting the performance of a Natural Language Processing (NLP) model. These word embeddings come in handy during hackathons and of course , in real world problems as well.

But why should we learn our own embeddings ? The number of Trainable Parameters increases while learning embeddings from scratch. This results in a slower training process. Learning embeddings from scratch might also leave you in

an unclear state about the representation of the words.

What are the different pretrained word embeddings ? The embeddings broadly classified in to 2 classes :- Word level and character level embeddings. ELMO and Flair embeddings are examples of character level embeddings. For now we are going to cover two popular word level pre trained word embeddings.

- ✓ - Google's Word2Vec
- ✓ - Stanford's Glove

Word2Vec is one of the most popular pre trained word embeddings developed by Google. Word2Vec is trained on the Google News dataset (about 100 billion words) It has several use cases such as Recommendation Engines , Knowledge discovery , and also applied in the different Text Classification problems. The architecture of Word2Vec is really simple. It is a feed forward neural network with just one hidden layer. Hence , it is sometimes referred to as a shallow neural network architecture.

Depending on the way the embedding are learned word2vec are classified in to two approaches.

- ✓ - Continuous Bag Of Words (CBOW)
- ✓ - Skip - gram model

Continuous Bag Of Words (CBOW) model learns the focus word given the neighbouring words whereas the skip-gram model learns the neighboring words given the focus word. That's why , continuous Bag Of Words and Skip gram are inverses of each other.

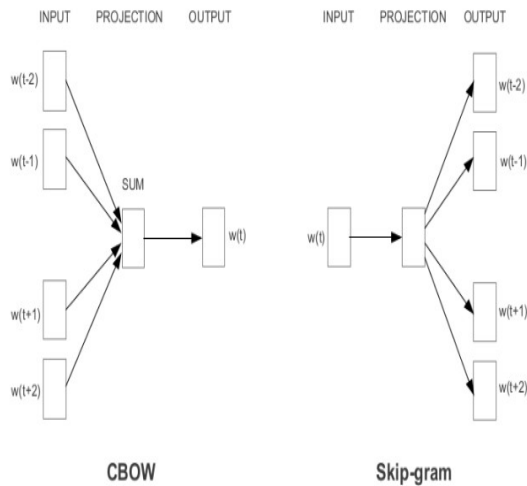
For example , consider the sentence : "I have failed at times but i never stopped trying" Let's say we want to learn the embedding of the word "failed" , so here the focus word is "failed".

The first step is to define a context window , a context window refers to the number of words appearing on the left and right of a focus word. The words appearing in the context window are known as neighboring words or context words. Let's fix the context window to 2 and then input and output pairs for both approaches.

- Continuous Bag Of Words :- Input = [I , have , at , times ] , output = failed

- Skip Gram :- Input = failed , output = [I , have , at , times ]

As you can see here , CBOW accepts multiple words as input and produces a single word as output where as skip gram accepts a single word as input and produces multiple word as output.



Hugging Face is a community and data science platform that provides tools that enable users to build , train and deploy ML models based open source code and technologies. It's a place where a broad community of data scientists , researchers and ML engineers can come together and share ideas , get support and contribute to open source projects.

Hugging Face addresses this need by providing a community “Hub”. It's a central place where anyone can share and explore models and datasets. They want to become a place with the largest collection of models and datasets with the Goal of democratizing AI for all.

Hugging Face Transformers provides APIs and tools to easily download and train state of the art pre-trained models. The hugging Face Transformers package is an immensely popular python library providing pretrained models that are extra ordinarily useful for a variety of natural language processing (NLP) tasks. It previously supported only PyTorch but as of late 2019 , TensorFlow2 is supported as well.

# MACHINE TRANSLATION USING ENCODER DECODER FRAMEWORK

Prior to transformers , recurrent architectures such as LSTMs were the state of the art in NLP. These architectures contain a feed back loop in the network connections that allow information to propagate from one step to another , making them ideal for modeling sequential data like text.

An RNN receives some input (which could be a word or character) , feeds it through the network and outputs a vector called the hidden state. At the same time , the model feeds some information back to itself through the feedback loop , which it can then use in the next step. These can be more clearly seen if we “unroll” the loop as shown on the right side of the figure. The RNN passes information about its state at each step to next operation in the sequence. This allows an RNN to keep track of information from previous steps and use it for its output predictions.

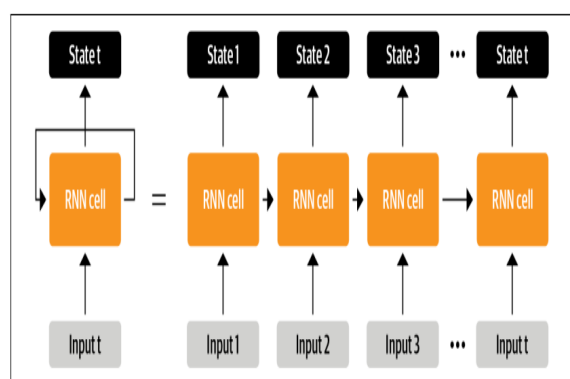


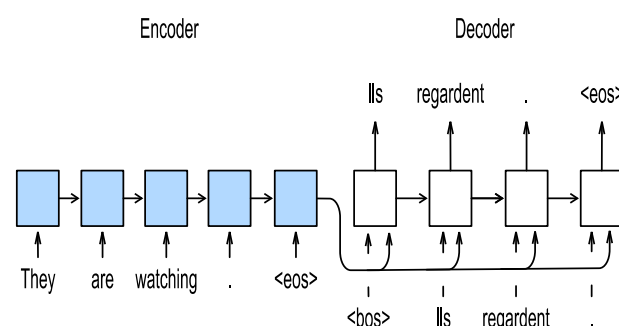
Figure 1-2. Unrolling an RNN in time

These architectures were (continue to be) widely used for NLP tasks , speech processing and time series.

One area where RNN’s played an important role was in the development of machine translation systems , where the objective is to map a sequence of words in one language to another. This kind of task is usually tackled with an encoder – decoder or sequence to sequence architecture , which is well suited for situations where the input and output are both sequences of arbitrary length. The job of the encoder is to encode the information from the input sequence in to a numerical representation that is often called the last hidden state. This state is then passed to the decoder , which generates the output sequence one at a time.

Although elegant in its simplicity , one weakness of this architecture is that the final hidden state of the encoder creates an information bottleneck : it has to represent the meaning of the whole input sequence in one long vector and this is all the decoder has access to when generating

the output. This is especially challenging for long sentences , where information at the start of the sequence might be lost in the process of compressing everything to a single , fixed representation.



Fortunately , there is a way out of this bottleneck by allowing the decoder to have access to all the encoder’s hidden states. The general mechanism for this is called attention , and it is a key component in many modern neural network architectures. Understanding how attention was developed for RNN’s will put us in good shape to understand one of the main building blocks of the Transformer architecture.

## ATTENTION MECHANISM

The main idea behind attention is that instead of producing a single hidden state for the entire input sequence , the encoder outputs a hidden state at each step that the decoder can access. However , using all the states at the same time would create a huge input for the decoder , so some mechanism is needed to prioritize which states to use. This is where attention comes in , it lets the decoder to assign a different amount of weight or “attention” to each of the encoder states at every decoding time step.

By focusing on which input tokens are most relevant at each time step , these attention based models are able to learn non-trivial alignments between the words in a generated translation and those in a source sentence. This figure below visualizes the attention weights for an english to french translation model , where each pixel denotes weight.

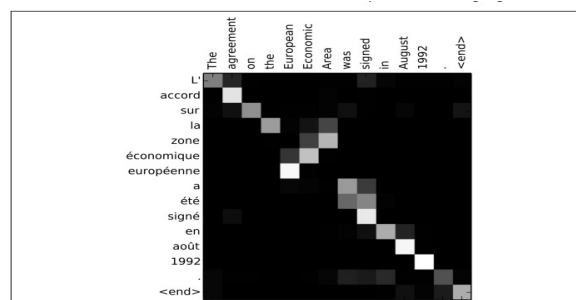


Figure 1-5. RNN encoder-decoder alignment of words in English and the generated translation in French (courtesy of Dzmitry Bahdanau)

Although attention enabled the production of much better translations, there was still a major shortcoming with using recurrent models for the encoder and decoder, the computations are inherently sequential and can not be parallelized across the input sequence.

## DEEP DIVE IN TO ATTENTION MECHANISMS

Attention is a mechanism that was developed to improve the performance of the encoder decoder RNN on machine translation. The model is comprised of two sub models, which is encoder and decoder.

Encoder :- The encoder is responsible for stepping through the input time steps and encode the entire sequence in to a fixed length vector called a context vector. Decoder is responsible for stepping through the output time steps reading from the context vector. Attention model is a neural extension of their previous work on the encoder decoder model.

Attention is proposed as a solution to the limitation of the encoder decoder model, encoding the input sentences to one fixed length vector from which to decode each output time step. This issue is believed to be more a problem when decoding long sentences. A potential issue with this encoder decoder approach is that a neural network needs to be able to compress all of the necessary information of a source sentence in to a fixed length vector. This may make it difficult for the neural network to cope with long sentences, especially that are longer than sentences in the training corpus.

Attention is proposed as a method to both align and translate. Alignment is the problem of machine translation that identifies which parts of the input sequence are relevant to each word in the output, where as translation is the process of using the relevant information to select the appropriate output.

We introduce an extension to the encoder and decoder model which learns to align and translate jointly. Each time the proposed model generates a word in a translation, it searches for a set of positions in a source sentence where the most relevant information is concentrated. The model then predicts a target model based on the context vectors and all the previous generated target words.

Instead of encoding the input sequence in to a single fixed context vector, the attention model develops a vector that is filtered specifically for each output time step. As with the encoder decoder paper the technique is applied to a machine translation problem and uses GRU units rather than LSTM memory cells. In these case bi-directional input is used where the input sequences are provided both forward and backward which are then concatenated before being passed on the decoder.

## THE PROBLEM OF LONG SENTENCES

How does the attention model makes us solve the problem of long sentences ?

We've been using Encoder and Decoder architecture for machine translation, one Recurrent Neural Networks reads the sentence and then the different one outputs a sentence, there is a modification to this called the Attention Model that make all this better.

So given a very much long french sentence like :-

“Jane s'est rendue en Afrique en septembre dernier a appercia et a rencontre beaucoup de gens merveilleux ; elle est revenue en parlent comment son voyage etait merveilleux, et elle me tente d'y aller aussi”.

Now what we are asking the encoder neural network to do is to read the whole sentence and then memorize the whole sentence and then store it to the activations and then the decoder neural network then generate the english translation.

Now the way human translator would translate the sentence is not to first read the whole french sentence and memorize the whole thing and then regurgitate an english statement from scratch but instead what a human translator would do is read the first part of it may be generate part of the translation and look the second part, generate a few more words and so on. We kinda work part by part through the sentence because it is just really difficult to memorize the whole long sentence, and so what we see for the encoder decoder architecture is that it works quite well for short sentences but for every long sentences may be longer than 30 or 40 words, the performance comes down.

## FORMALIZE THE ATTENTION MODEL INTUITION

Let's illustrate this with a short sentence, even though the ideas is developed for longer sentences. Let's use a bi-directional RNN in order to input compute some set of features for each of the input words. Here we drawn the standard bidirectional recurrent neural network with the outputs  $y_{<1>}$  ...  $y_{<5>}$  but now we are not going to do a word to word translation so we will get rid of the  $y$ 's on the top.

But using a Bi-Directional Recurrent Neural Network what we have done is for each of the words (five position words) in the sentence we can compute a very rich set of features about the word in the sentence or may be surrounding words.

We're going to use another recurrent neural network to generate the english translation and instead of using “a” to denote the activation, in order to avoid confusion with the activation we are going to use different notations, let's call it “s”.

Now the question is when we are trying to generate the first word (“Jane”), what part of the french input sentence should be looking at ? So what the attention model would be computing is a set of attention weights and we're going to use  $\alpha_{<1,1>}$  to denote to generate the first word, how much should pay attention to the piece of

information that we get from the first word  $x_{<1>}$  from our feature vector (our activation).

And we also come up with a second attention weights, let's call it  $\alpha$  (1,2) which tells us while we are trying to compute the first word "Jane" how much attention should be paying to this second word from the input and so on and together this will tell us, what is exactly the context  $c[1]$  we should be paying attention to and that is input to our first recurrent neural network unit and then try to generate the first word so this is one step of the RNN.

Question :-

- ✓ How exactly does this context defined ?
- ✓ How does we compute this attention weight  $\alpha$  ?
- ✓ How does the attention model solves the problem of the encoder and decoder, in the translation of longer sentences.
- ✓ What is the work of the second neural network "s" ?

The  $\alpha_{<t,t'>}$  allows it on every time step to look only may be in a local window of the french sentence to pay attention to when generating a specific english word.

Now, let's formalize the attention model. We have an input sentence and we use a Bi-Directional recurrent neural network but we can also use a Bi-Directional GRU or a Bi-Directional LSTM to compute features on every word.

And for the forward recurrence we would have  $a$ (forward) and  $a$ (backward) for the backward recurrence. Technically like  $a_{<0>}$  in the forward step we have also  $a_{<6>}$  for the backward step which is a vector of zeros and at every time step even though we have the features computed from the forward recurrence and from the backward recurrence in the bi-directional RNN ( $a_{<0>}$  forward,  $a_{<0>}$  backward) and we are going to use  $a_{<t'>}$  for both of these concatenated together.

$$a_{<t'>} = (a_{<t'>} \text{ forward} + a_{<t'>} \text{ backward})$$

We are saying  $t'$  to denote that it is a french sentence and this is going to be a feature vector for the time step  $t'$  in both forward and backward direction. So the way we define the context is actually to sum the features from different time steps weighted by this attention weights.

So more formally the attention weights will satisfy this

$$\sum_{t'} \alpha_{<t,t'>}$$

The  $\alpha$  must be non-negative and they are summed to 1 and we will have the context at time step 1 like this :-  
 $C[1] = \sum_{t'} \alpha_{<t,t'>} a_{<t'>}$

The context is going to be the sum over  $t'$ , weighted sum of the activation function with the attention weights. Where the activation function  $a_{<t'>} = (a_{<t'>} \text{ forward}, a_{<t'>} \text{ backward})$ . In other word when we are generating the output word how much should we paying attention to the  $t'$  input word.

So using the context vector the above network "s" looks like a standard recurrent neural network with the context vector as output and we can just generate the translation one word at a time.

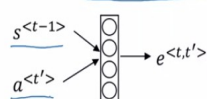
## HOW DO WE CALCULATE THE ATTENTION WEIGHTS.

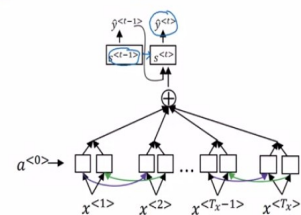
So the only remaining thing to do is to define how to actually compute these attention weights. So just to recap,  $\alpha(t, t')$  is the amount of attention we should paid to  $a_{<t'>}$ , when we are trying to generate the  $t$ th words in the output translation.

$\alpha_{<t,t'>}$  = amount of attention  $y_{<t>}$  should pay to  $a_{<t'>}$ , so the formula we could use to compute  $\alpha_{<t,t'>}$  is

### Computing attention $\alpha_{<t,t'>}$

$\alpha_{<t,t'>}$  = amount of attention  $y_{<t>}$  should pay to  $a_{<t'>}$

$$\alpha_{<t,t'>} = \frac{\exp(e_{<t,t'>})}{\sum_{t'=1}^T \exp(e_{<t,t'>})}$$




This is essentially a softmax, to make sure that these weights sum to one if we sum over  $t'$ . Now how do we compute the attention weights? Well one way to do so is to use a small neural network. So  $s_{<t-1>}$  was the neural network from the previous time step. So if we are generating  $y_{<t>}$  then  $s_{<t-1>}$  was the hidden state from the previous time step that just fall in to  $s_{<t>}$  and that is one input to very small neural network, usually one hidden layer in neural network because we need to compute these a lot and then  $a_{<t'>}$  the features from time  $t'$  is the other inputs and the intuition is, if we want to decide how much attention to pay to the activation of  $t'$ , well the things that will depend the most on is what our hidden state activation from the previous time step. We don't have the current state of activation yet because of context feeds in to this so we haven't computed that but look at whatever our hidden translation and then for each of the positions, so it seems pretty natural that  $\alpha_{<t,t'>}$  and  $e_{<t,t'>}$  should depend on these two quantities.

But we don't know what the function is, so one thing we could do is just train whatever this function should be and trust back propagation, trust gradient descent to learn the right function.



This neural network does a pretty decent job telling us how much attention  $y_{<t>}$  should pay to  $a_{<t>}$  and this formula makes sure that the attention weights sum to one and then as we chug along generating one word at a time, this neural network actually pays attention to the right part of the input sentence that learns all this automatically using gradient descent.

## ATTENTION MODELS FOR IMAGE CAPTION

How does the attention models enables us for the image captioning purpose. These also applied to other problems as well as such as Image Captioning, the task is to look at the picture and write any caption for that picture.

We could have a very similar architecture, look the picture and pay attention only to part of the picture at a time while you're writing a caption for the picture.

## THE PROBLEM WITH ATTENTION MECHANISM

Although attention enabled the production of much better translations, there was still a major shortcoming with using recurrent models for the encoder and decoder: the computations are inherently sequential and can not be parallelized across the input sequence.

## SELF ATTENTION MECHANISM

- ✓ What is self attention ?
- ✓ What does the term self represents ?
- ✓ What makes the self attention different from original attention mechanism ?
- ✓ How is self attention embedded in the transformer network ?
- ✓ How does the parallel computing works ?

As the complexity of our sequence task increases, so does the complexity of our model. We have started with RNN, found that it had some problems with vanishing gradients, which made it hard to capture long range dependencies and sequences. We then looked at the GRU and then LSTM model as a way to resolve many of those problems where we may use gates of control the flow of information while these models improve to control the flow of information, they also came with increased complexity.

So as we move from our RNN's to GRU to LSTM the models became more complex and all of these models are still sequential models in that they ingested may be the input sentence one word at the time and so, as if each unit was like a bottleneck to the flow of information because to compute the final unit for example we first have to compute the outputs of all the units that came before. So for this we will learn a transformer architecture, which allows us to run a lot more of these computations for the entire sequence in parallel, so we can ingest an

entire sentence all at the same time rather than just processing it one word at a time from left to right.

So what we see in the attention network is a way of computing a very rich, a very useful representation of words but with something more akin to the style of parallel processing.

To understand transformer network, we must first see two basic concepts :-

- ✓ Self attention
- ✓ Multi-Head Attention

The goal of self attention is, if we have say a sentence of five words, we will end up computing five rich representations for this five words, was going to write  $A_{<1>}$ ,  $A_{<2>}$ ,  $A_{<3>}$ ,  $A_{<4>}$ ,  $A_{<5>}$  and this will be an attention based way of computing representations for all of the words in our sentence in parallel.

Then the multi head attention is basically a for loop over the self attention process so we need up with a multiple versions of these representation and it turns out these representations can be used for machine translations or other NLP task to create effectiveness. So first let's see self attention which provides a very rich representation of the words.

We've seen how attention model is used with sequential neural networks such as RNN's. To use attention with a style more like CNNs, we need to calculate self attention, where we create attention based representations for each of the words in our sentence.

$A(q, k, v)$  = attention based vector representation of a word.

Let's use our running example :-

$X_{<1>}$     $X_{<2>}$     $X_{<3>}$     $X_{<4>}$     $X_{<5>}$

Jane   Visite   l'Afrique   en   Septembre

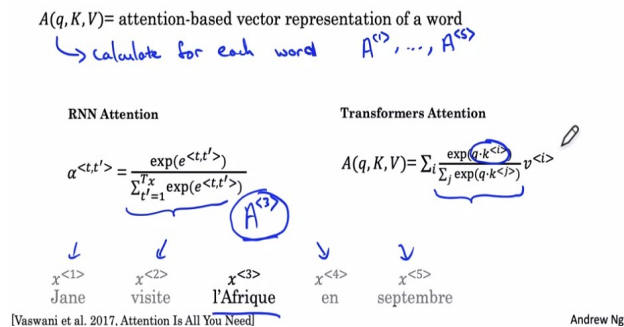
Our goal will be for each word to compute an attention based representation like this, so we will end up with five of this, since our sentences has five words ( $A_{<1>}$ ,  $A_{<2>}$ ,  $A_{<3>}$ ,  $A_{<4>}$ ,  $A_{<5>}$ ).

The running example we are going to use is take the word l'Afrique in this sentence then we will step through on how the transformer network's self attention mechanism allows us to compute  $A_{<3>}$  for this word and then the same thing for other words in the sentence as well. Now as we know, one way to represent l'Afrique would be to just look up the word embedding for l'Afrique or Africa as a site of historical interests or as a daily destination or as a world's second largest content.

Depending on how we're thinking l'Afrique we may choose to represent  $A_{<3>}$  will do. It will look at the surrounding words to try to figure out what's actually

going on in how we're talking about Africa in this sentence and find the most appropriate representation for this. In terms of the actual attention calculation, it won't be too different from the attention mechanism formula as applied to the context of RNN's except we will compute these representations in parallel for all five words in a sentence.

## Self-Attention Intuition



The main difference is that in every word we have three values called the query, key and value, these vectors are the key inputs to computing the attention value for each word.

Our first step is we are going to associate each of the words with values called query, key and value pairs. We are going to see how well the queries and keys match, but all of them (q, k, v) are vectors.

Query(q)	key(k)	Value(v)
$q_{<1>}$	$k_{<1>}$	$v_{<1>}$
$q_{<2>}$	$k_{<2>}$	$v_{<2>}$
$q_{<3>}$	$k_{<3>}$	$v_{<3>}$
$q_{<4>}$	$k_{<4>}$	$v_{<4>}$
$q_{<5>}$	$k_{<5>}$	$v_{<5>}$

**What does the q, k and v represents in the self attention mechanism of the Transformer architecture ?**

In the self attention mechanism of the Transformer architecture q, k, and v are known as the query, key and value vectors respectively.

- The query vector (q) represents the vector of the current input word/token in the self attention mechanism.

- The key vector (k) represents the vector of all other words / token in the same sequence, which are used to compute the attention weights.

- The value vector (v) represents the vector that is used to weight the importance of each key based on its relevance to the current input word/token.

**So what is the difference between the key and the value vector ?**

The key vector and the value vector in the self attention mechanism of the Transformer architecture serve different purposes and have different roles.

\* The key vector is used to represent the relationship between the current input word/token (represented by the query vector) and all the other words/tokens in the input sequence. The dot product between the query and key vectors produces an attention score that measures the relevance of each key vector to the current word/token.

- In other words, the key vector is used to encode the context or positional information of each token in the sequence. The vector or the content of the key vector is determined by the model during training, based on the patterns in the input data. Each key vector is learned to capture the semantic and syntactic relationships between the current input word / token and all the other words/tokens in the input sequence. For example, in natural language processing tasks, the key vectors may represent the word in a sentence, and the model may learn to encode information such as the relative position of each word to others, the grammatical role of each word in the sentence and the semantic relationship between the words.

\* The value vector, on the other hand, contains the information that the model needs to learn and use to make predictions. In other words, the value vector represents the content associated with the corresponding key vector.

The attention weights are computed by taking the dot product of the query vector with all the key vectors, and then normalizing the resulting scores using the softmax function. These weights are then used to weight the corresponding value vectors, which are summed up to obtain the output of the self attention mechanism.

The self attention is used in the Transformer architecture to allow the model to focus on different parts of the input sequence at different positions, enabling it to capture long-range dependencies and improve the quality of the model's predictions.

If  $X_{<3>}$  is the word embedding for l'Afrique, the way this vectors is computed is a learned matrix.

$$Q_{<3>} = W_q X_{<3>}$$

$$K_{<3>} = W_k X_{<3>}$$

$$V_{<3>} = W_v X_{<3>}$$

These matrices  $W_q$ ,  $W_k$ ,  $W_v$  are parameters of this learning algorithm and they allow you to pull these query, key and value vectors for each word but what are these query, key and value. We can think them as a loose

analogy to databases where we can have queries and key – value pairs.

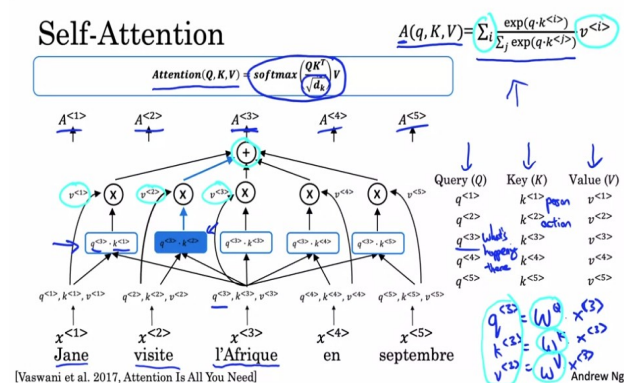
X<1> X<2> X<3> X<4> X<5>  
Jane Visite l'Afrique en Septembre

q<3> is a question that we get to ask about l'Afrique , so q<3> may represent like , what is happening there ? Is it a destination ? So what we are going to do is compute the inner product between q<3> and k<1> and this will tell us how good is an answer to the question of what's happening in Africa and then we will compute the inner product between q<3> and k<2> and this intended to tell us how good is "visite" an answer to the question of "what Is happening in Africa" and so on for the other words in the sequence and the goal of this operation is to pull the most information that is needed to help us compute the most useful representation A<3>.

So if K<1> represents that this word is a person because Jane is a person and K<2> represents that the second word , visite is an action , then we may find that k<2> has the largest value and this intuitive example might suggest that visite gives more relevant context for what's happening Africa , which is it's viewed as a destination for a visit.

So what we will do is take five values (q<3>k<1> , q<3>k<3> , q<3>k<4> , q<3>k<5>) in and compute the softmax over them and in our example q<3>k<2> corresponding to word visite may be the largest value (that is why we are making that part of the word bold).

Now after computing the softmax , we are going to multiply with the value vector for each word and then finally we sum it up and so all of these values will give us A<3>.



Now after computing the softmax , we are going to multiply with the value vector for each word and then finally we sum it up , and why are doing that ?

After computing the softmax over the dot products between the query and key vectors , the attention mechanism weights corresponding value vectors and sums them up to obtain the final output of the self attention mechanism. This is done to compute a weighted average of the values , where the weights are given by the attention scores (The weights are given by the attention

scores , by weighted summing it with the value vector we will obtain the output of the self attention mechanism)

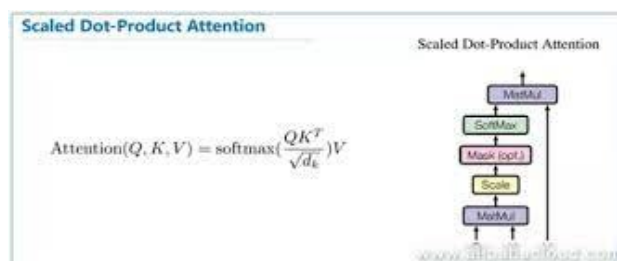
The intuition behind this operation is that the attention scores tell us how much importance the model should place on each value vector when computing the output. By weighting the value vectors with their corresponding attention scores and summing them up , we obtain a representation of the input that is focused on the most relevant parts of the sequence.

In other words , the attention mechanism allows the model to selectively attend to that parts of the input that are most informative for the task at hand. For example :- In natural language processing tasks , the attention mechanism allows the model to focus on the most relevant words in the input sentence when generating a translation or predicting the next word in sequence.

The output of the self attention mechanism is then passed through a feed forward neural network , which applies a non-linear transformation to the output before passing it on to the next layer in the network. The feed forward network helps to capture more complex patterns in the data and improve the performance of the model on the task at hand.

So another way to write A<3> is really as A(q<3> , k , v) and the key advantage of this representation is the word l'Afrique isn't some fixed word embedding but instead , It let's the self attention mechanism realize that l'Afrique is the destination of a visite and this is a richer and more useful representation for this word.

Now we have been using the third word , l'Afrique as a running example but we could uses this process for all the five words in our sequence to get similarity rich representations for Jane , l'Afrique , en , septembre. If we put all of these five computations together , denotation used in literature looks like this , where we can summarize all of these computations that we just talked about for all the words in the sequence by writing attention (Q , K , V) where Q , K , V matrices with all of these values , and this is just a compressed or vectorized representation of our equation.



The term in the denominator is just to scale the dot product , so It does not explode , so another name for this type of attention is the scaled dot product attention.

The importance of the denominator dk lies in its ability to control the magnitude of the dot product between Q and K , which in turn affects the scale of the resulting attention weights.

If the dot product between Q and K is large, the softmax function in the formula may saturate and result in very small gradients during back propagation. However, by scaling the dot product by the square root of the dimension of K (I.e.,  $d_k$ ), we can control the scale of the resulting weights, preventing the softmax function from saturating.

To illustrate more, scaling the softmax by the square root of the dimension of K, will ensure that the dot products between query and a key, don't grow too large (because if it grows too large the gradient of the softmax will be too small, this is because softmax has a graph like sigmoid which leads to saturation when the matrix multiplication is too large), so  $\sqrt{d_k}$  is a scaling factor.

**Question :- Why do we scale it using  $d_k$ ? And why are we squaring it?**

scaling the dot product by  $d_k$  has been shown to help stabilize the training process and improve the performance of the self-attention mechanism, especially when dealing with long sequences. This is because the magnitude of the dot product can grow large as the dimensionality of the input increases, and scaling by  $d_k$  helps to mitigate this effect.

We scale the dot product between Q and K by the square root of the dimension of K (i.e.,  $d_k$ ) for two main reasons:

1) Control the Magnitude of the Dot Product: The dot product of two vectors is a function of the magnitudes of the vectors and the angle between them. As the dimensionality of the vectors grows, so does the magnitude of the dot product. This can result in numerical instability and poor performance when the dot product is used as a similarity metric. By dividing the dot product by the square root of the dimension of K, we effectively scale it down to control its magnitude and stabilize the training process.

2) Ensure Consistency in Softmax Scaling: The softmax function is used to convert the dot products into a probability distribution over the values in V. The magnitude of the dot products can affect the scale of the resulting probabilities. By dividing the dot product by the square root of the dimension of K, we ensure that the scale of the probabilities is consistent across different values of K. This helps the model to focus on the most relevant parts of the input sequence.

We square the denominator  $d_k$  to make the scaling more aggressive. This is because we want to emphasize the importance of the scaling factor in controlling the magnitude of the dot product. By squaring  $d_k$ , we effectively double its effect on the dot product, making the scaling more aggressive and helping to further stabilize the training process.

Overall, scaling the dot product by the square root of the dimension of K and squaring it is a simple but effective technique for controlling the magnitude of the dot product and stabilizing the self-attention mechanism during training.

To recap, associated with each of the five words we end up with a query, key and value. The query lets you ask a question about that word, such that what is happening in Africa. The key looks at all of the other words and by similarity to the query, helps you figure out which words give the most relevant answer to that question.

In our case, Visite is what's happening in Africa, someone is visiting Africa. Then finally, the value allows the representation to plug in how visite should be represented with in A<3> with in the representation of Africa. This allows to come up with a representation for the word Africa that says this is Africa and some one is visiting Africa.

This is much more nuanced and much more representation for the words than if we just had to pull up the same fixed word embedding for every single word without being able to adapt it based on what words are the left and to the right of that word.

## THE DIMENSION OF THE SELF ATTENTION MECHANISM

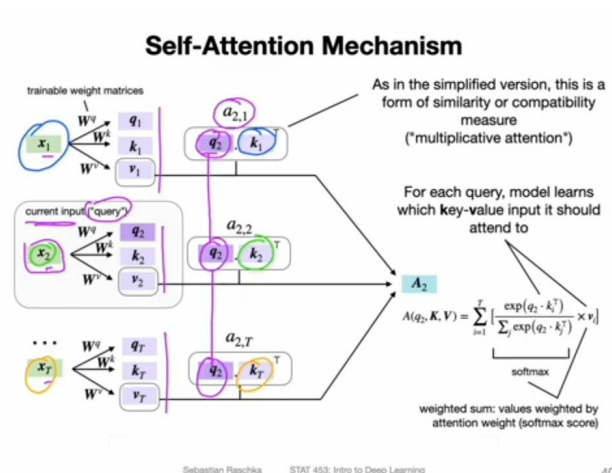
We are now adding 3 trainable weight matrices that are multiplied with the input sequence embeddings ( $X_i$ 's)

$$\text{Query} = W_q X_i$$

$$\text{Key} = W_k X_i$$

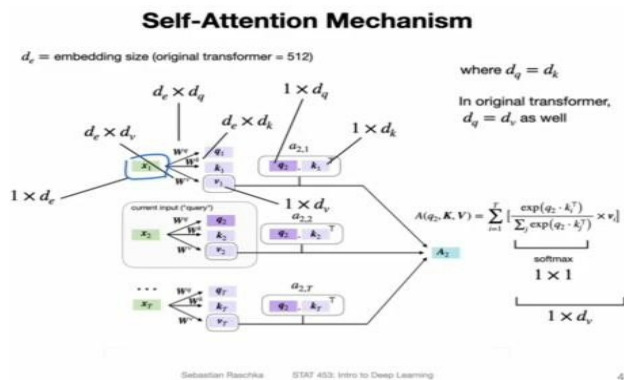
$$\text{Value} = W_v X_i$$

So here is the self attention mechanism looks for a particular input.

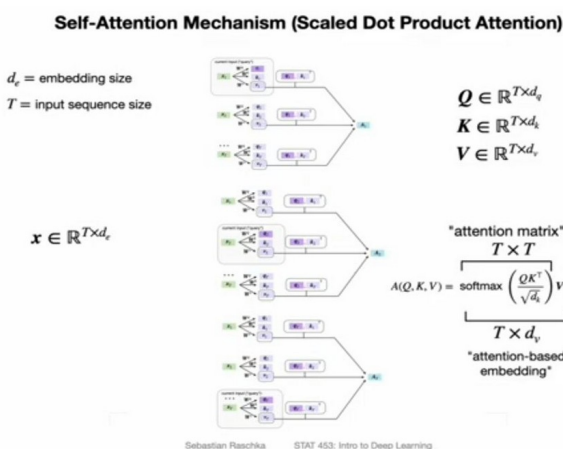


For each query, the model learns which key value it should attend to





1\*d<sub>e</sub> is the embedding size , in the original paper they used 512 embedding size.

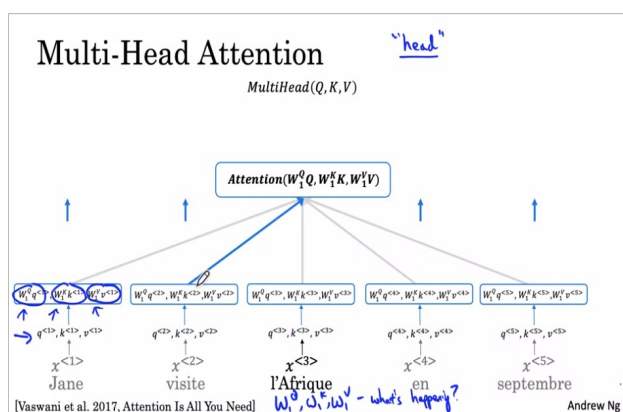


The weight matrices must have a row which is the same as the column of the embedding matrix. d<sub>q</sub> = d<sub>k</sub> they must be in the same dimension for the dot product.

## WHAT IS THE DOWN SIDE OF ATTENTION MODELS

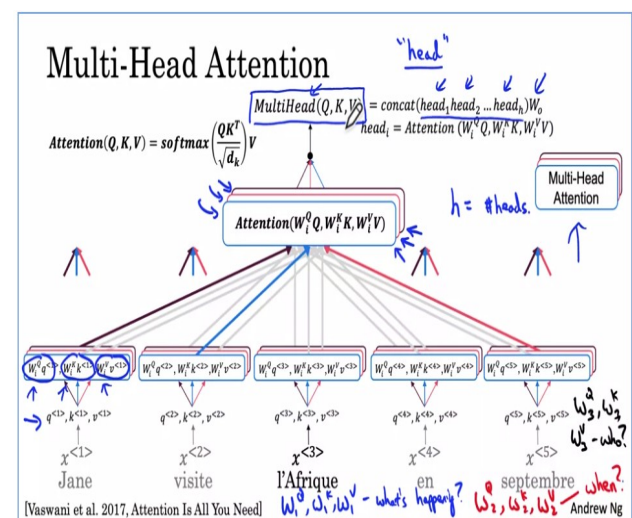
Now one downside of these algorithm is that , it does take a quadratic cost to run this algorithm , if we have T<sub>x</sub> word inputs and T<sub>y</sub> words in the output then the total number of these attention parameters is gonna be T<sub>x</sub> times T<sub>y</sub> , therefore these algorithm run in a quadratic.

## MULTI – HEAD ATTENTION MECHANISM



Multi head attention is basically a big for loop over the self attention mechanism that we learned before , and each time we calculate the self attention for a sequence is called a head and the name multi head attention refers to a normal self attention but a bunch of times.

Remember that we got the vectors Q , K and V for each of the input terms multiplying them by a few matrices W<sub>q</sub> , W<sub>k</sub> , W<sub>v</sub>. So with the computation the word visite gives the best answer to the query “what’s happening” which is why we highlighted with the blue arrow to represent that the inner product between the key for l’Afrique has the highest value with query for visite. So this is how we get the representation for l’Afrique and we do the same for Jane , visite and for other words. So we end up with five vectors to represent the five words in the sequence. So this is a computation to carry out for the first of the several heads we use in multi head attention. And so we would step through exactly the same calculation that we had just now for l’Afrique and for other words and end up with the same attention values , A<1> through A<5> , but now we’re going to do this not once but a handful of times. So rather than having one head , we may now have eight heads , which just means performing this whole calculation may be eight times.



Now let’s do this computation with the second head , the second head will have a new set of matrices , we are going to write W<sub>q2</sub> , W<sub>k2</sub> , W<sub>v2</sub> , that allows this mechanism to ask and answer a second question. So the first question was “what’s happening” may be the second question is “when something is happening” and so instead of having w1 here , in general case we will have w<sub>i</sub> and we’ve now stacked up the second head behind the first one , the one in the red. So we repeat a computation that is exactly the same as the first one but with this new set of matrices instead and we end up with in these may be the inner product between the September key and the l’Afrique query will have the highest inner product , so we are going to highlight by a red arrow to indicate that the value for September will play a larger role in this second part of the representation for l’Afrique.

May be the third question we now want to ask as represented by w<sub>q3</sub> , w<sub>k3</sub> , w<sub>v3</sub> is , who has something to



do with Africa ? And in this case when we do this computation for the third time , may be the inner product between Jane's key vector and the l'Afrique query vector will be the highest and self highlighted this is a black arrow. So that Jane's value will have the greatest weight in this representation which we have stacked on at the back. In the literature , the number of heads is usually represented by the lower case H , and so H is equal to the number of heads.

And we can think of each of these heads as a different feature , and when we pass these features a new network we can calculate a very rich representation of the sentence. Calculating this computations for the three heads or the eight heads or whatever the number , the concatenation of these three value or 8 values is used to compute the output of the multi head attention and so the final value is the concatenation of all of these h heads and then finally multiplied by a matrix W0.

So doing the self attention multiple times , we now understand the multi head attention mechanism , which let us ask multiple questions for every single word and learn a much richer and much better representation for every word.

**Question :- Why are we multiplying the concatenated attentions with a weight W0 ?**

In multi-head self attention , the input sequence is transformed by projecting it in to multiple subspaces. Each of these subspaces then undergoes a separate self attention mechanism , generating a set of self attention outputs for each subspace. These attention outputs are then concatenated and passed through a linear transformation (represented by weight matrix W0) to generate to the final output of the multi-head self attention layer.

The reason we multiply the concatenated attentions with weight matrix W0 is to learn a linear transformation that maps the concatenated attention outputs to the desired output dimensionality. Essentially , this linear transformation allows the model to learn how to combine the information from multiple attention heads to produce an output that best represents the input sequence.

**How does multiplying the concatenated attentions by a weight matrix W0 creates a linear transformations ?**

Multiplying the concatenated attentions by a weight matrix W0 is a linear transformation because it is a linear combination of the input features (the concatenated attention outputs ) and the learnable parameters of the weight matrix. **What does it mean by linear transformation deeply ?**

A linear transformation is a mathematical operation that maps one vector space to another in a way that preserves the basic properties of the original space.

Let's say that the concatenated attention outputs have a shape (batch\_size , sequence\_length ,

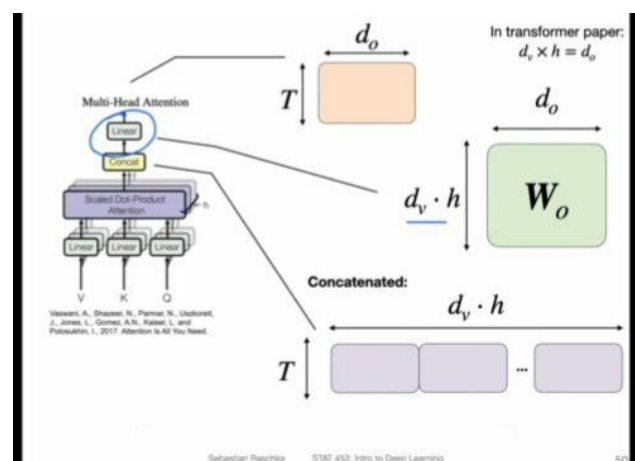
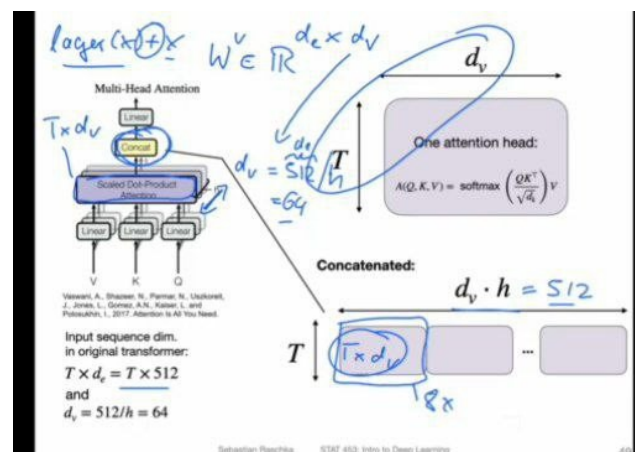
num\_attention\_heads \* attention\_head\_size) , where num\_attention\_heads is the number of attention heads and attention\_head\_size is the dimensionality of each attention head output. The weight matrix W0 has a shape of (num\_attention\_heads \* attention\_head\_size , output\_dimension ) , where output\_dimension is the desired dimensionality of the final output.

To compute the linear transformation , we first reshape the concatenated attention outputs in to a 2D tensor of shape (batch\_size \* sequence\_length , num\_attention\_heads \* attention\_head\_size) , then we multiply this tensor by the weight matrix W0 , resulting in a tensor of shape (batch\_size \* sequence\_length , output\_dimension)

Finally , we reshape this tensor back to the original shape of (batch\_size , sequence\_length , output\_dimension) to obtain the final output of the multi-head self attention layer. This linear transformation can be though of as a weighted sum of the input features , where the weights are learned by the model during training.

In summary , multiplying the concatenated attention outputs by a weight matrix W0 creates a linear transformation because it is a linear combination of the input features and learnable parameters of the weight matrix.

## DIMENSION OF THE MULTI HEAD ATTENTION



# The Transformer Architecture

- ✓ What is the transformer architecture ?
- ✓ Why do we need it ?
- ✓ How does the transformer architecture works ?
- ✓ What feature of transformer change the world of NLP by storm ?
- ✓ How to implement it using PyTorch

We already familiarized ourselves with the concept of self attention as implemented by the transformer attention mechanism for neural machine translation. We will now be shifting our focus to the details of the Transformer Architecture itself to discover how self attention can be implemented without relying on the use of recurrence and convolution.

**Question :- A Transformer architecture without relying on recurrence behavior of self attention makes sense but how does removing the relying behavior on convolution will be helpful for the Transformer Architecture , what was the limitation of having a convolution behavior from the first place ?**

The NLP transformer architecture is not relying on convolutional behavior because it is designed to address a different type of natural language processing problem than convolutional neural networks (CNNs). Convolutional neural networks are commonly used in computer vision tasks , where the input data is a 2D grid of pixels. Convolutional layers in a CNN operate on local regions of the input data and extract features that are shared across different regions. This works well for tasks such as image classification , where the spatial arrangement of the input data is important. In contrast , natural language processing tasks deal with sequential data , where the order of the input data is important. The transformer architecture was specifically designed for sequence to sequence tasks , such as machine translation , where the input and output sequences can have variable lengths. The transformer architecture uses self

attention to attend to different parts of the input sequence can have variable lengths. The transformer architecture uses self attention to attend different parts of the input sequence , allowing the model to learn long-range dependencies and capture relationships between words that are far apart in the input sequence. This is more effective approach for natural language processing tasks than convolutional neural networks.

**Question :- How does the transformer architecture learn long – range dependencies , capturing relationships between words that are far apart in the input sentence.**

The transformer architecture uses several mechanisms to learn long – range dependencies and capture relationships between words that are far apart in the input sentence.

1) **Self attention** :- The Transformer uses self – attention mechanism that allows each word in the input sentence to attend to all the other words in the sentence , regardless of their position. This mechanism enables the model to capture long -range dependencies by giving it the ability to focus on relevant information in the sentence , even if it is far away from the current word being processed.

2) **Multi – Head Attention** :- The Transformer also uses multi – head attention , which is a variation of self attention that allows the model to attend to different aspects of the input sentence at the same time.

3) **Positional Encoding** :- The Transformer uses positional encoding to inject information about the position of each word in the input sentence. This allows the model to differentiate between words that are far apart in the sentence and helps it learn long range dependencies.

4) **Layer Normalization** :- The Transformer uses layer normalization to normalize the activations of each layer in the model. This helps to reduce the impact of vanishing gradients and ensures that the model can capture long – range dependencies.

5) **Stacked Layers** :- The Transformer uses a stack of multiple layers , each with its own self attention and feed – forward layers. This enables the model to capture increasingly complex relationships between the words in the input sentence and helps it learn long range dependencies.

**Question :- How does layer normalizing the activation layer of the model reduce vanishing gradients ?**

Layer normalization helps to reduce the impact of vanishing gradients in deep neural networks , including the Transformer architecture , by ensuring that the activations of each layer are normalized and have a consistent distribution.

Vanishing gradients occur when the gradients propagated through the layers of a deep neural network become very small , making it difficult to update the parameters of the network through back-propagation. This can be a significant problem in deep models with many layers , as the gradients can become too small to update the weights of the lower layers effectively.

Layer normalization addresses this issue by normalizing the activation of each layer in the model. Specifically , It computes the mean and variance of the activations over the feature dimension and uses these statistics to

normalize the activations. This normalization process ensures that the activations of each layer have a consistent distribution regardless of the input distribution, which can help to stabilize the gradients.

By stabilizing the gradients, layer normalization makes it easier for the model to learn and update its parameters during training. It also helps to reduce the impact of vanishing gradients, as the normalized activations ensure that the gradients are not excessively small or large as they propagate through the layers.

In the Transformer architecture, layer normalization is applied after self attention and feed – forward layer, which helps to ensure that the activation of each layer have a consistent distribution and that the gradients are stable during training. This contributes to the Transformers ability to learn long – range dependencies and capture relationships between words that are far apart in the input sentence.

So what about Vision Transformer (ViT), if that is so ?

The Vision Transformer (ViT) is an adaption of the transformer architecture for computer vision tasks. The ViT applies the self attention mechanism of the Transformer to image patches, allowing the model to attend to different parts of the image and learn long – range dependencies between patches.

In the ViT, the input image is divided in to a grid on non – overlapping patches, which are then flattened and fed in to the Transformer encoder. Each patch is treated as a separate token, and the self – attention mechanism is used to compute the relationship between all pairs of patches. This allows the model to learn spatial relationships between different regions of the image, without relying on hand – crafted features or convolutional operations. The ViT has been shown to achieve state of the art results on a range of computer vision tasks, including image classification, object detection and semantic segmentation. By adapting the Transformer architecture to image data, the ViT provides a powerful and flexible approach to computer vision that can capture complex spatial relationships between different parts of an image.

The transformer architecture follows an encoder – decoder structure but does not rely on recurrence and

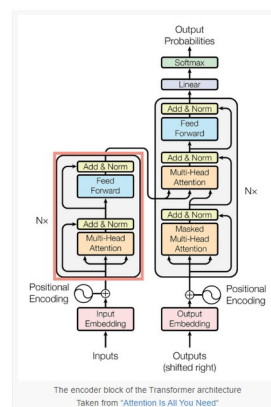
convolutions in order to generate an output. **How does the Transformer Architecture generates an output ?**

The task of the encoder, on the left of the transformer architecture, is to map an input sequence to sequence of continuous representations, which is then fed in to the decoder. **How does the encoder encodes in to a machine understandable representations ?**

The decoder, on the right half of the architecture, receives the output of the encoder together with the decoder output at the previous time step to generate an output sequence. **How does the decoder accepts the input from the encoder and adds it with the previous input.**

At each step of the model is auto – regressive, consuming the previously generated symbols as additional input when generating the next. **What does it mean by auto – regressive ?** A statistical model is auto regressive if it predicts future values based on past values.

The Encoder



## The Encoder Of The Transformer Architecture

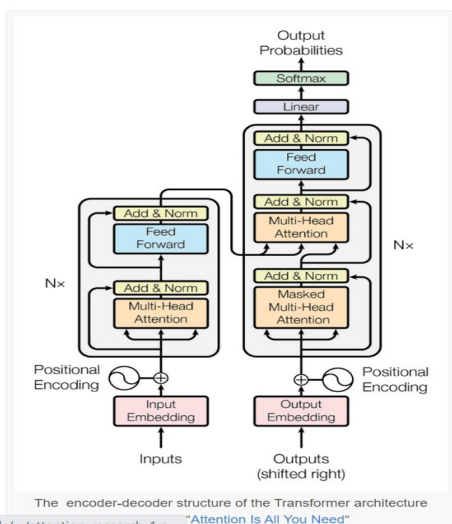
The encoder consists of a stack of  $N = 6$  identical layers, where each layer is composed of two layers.

1) The first sub layer implements a multi – head self attention mechanism. We have seen that the multi head mechanism implements  $h$  heads that receive a different linearly projected version of the queries, keys and values, each to produce  $h$  outputs in parallel that are then used to generate a final result.

2) The second layer is a fully connected feed forward network consisting of two linear transformation with Rectified Linear Unit (ReLU) activation in between.

$$\text{FFN}(x) = \text{ReLU}(w_1x + b_1)w_2 + b_2$$

**Question :- What is the advantage of this fully connected neural networks ?**



The Transformer architecture for natural language processing tasks includes a fully connected neural network in the encoder part, specifically in each Transformer layer. This fully connected network is known as the “ feed forward network ” and has several advantages.

1) **Non – linear transformation** :- The feed forward network applies a non – linear transformation to the output of the self – attention layer in each Transformer layer. This non – linearity allows the model to learn complex relationships between tokens in the input sequence, which can be critical for natural language processing tasks.

2) **Dimensionality Reduction** :- The feed forward network reduces the dimensionality of the output of the self – attention layer, which can help reduce the computational cost of the Transformer architecture. The self – attention layer outputs a matrix with dimensions corresponding to the number of tokens in the input sequence, which can be very large for long input sequences. The feed forward network reduces the dimensionality to a smaller, fixed size, which can be more computationally efficient to process in subsequent layers. **Prove this mathematically ?**

2) **Multi Modal Processing** :- The feed forward network can be used to incorporate other types of features in to the Transformer architecture, in addition to the token embeddings. For example, in some natural language processing tasks, it may be useful to incorporate information about the part of speech or name entity type of each token. The feed forward network can be used to process these additional features and integrate them in to the representation learned by the Transformer.

In summary, the feed forward network in the encoder of the Transformer architecture provides a non – linear transformation that allows the model to learn complex relationships between tokens, reduces the dimensionality of the input to improve computational efficiency, and can be used to incorporate other types of features in to the model. These advantages can help the Transformer architecture achieve state of the art performance on a wide range of natural language processing tasks.

The six layers of the transformer encoder apply the same linear transformations to all the words in the input sequence, but each layer employs different weight ( $w_1$ ,  $w_2$ ) and bias ( $b_1$ ,  $b_2$ ) parameters to do so.

**Question :- Why do we need this repeated six layers ?**

The Transformer architecture for natural language processing tasks includes both an encoder and decoder. The encoder is responsible for processing the input sequence, while the decoder generates the output sequence. The encoder uses a stack of  $N$  identical layers, where  $N$  is typically set to 6 or more. There are several reasons why a stack of multiple layers is needed in the encoder :-

1) **Capturing complex dependencies** :- Each layer in the encoder can capture different level of abstraction in the input sequence. The lower layers capture simple dependencies between nearby tokens, while the higher layers capture more complex and long – range dependencies. By stacking multiple layers, the model can capture increasingly complex relationships between tokens, allowing it to learn model sophisticated representations of the input sequence.

2) **Robustness to noise** :- Stacking multiple layers in the encoder can also help make the model more robust to noise and variations in the input sequence. Each layer can perform different type of processing on the input, allowing the model to learn multiple representations of the same input. This can help the model to identify and filter out noise in the input, making it more robust to variations and errors.

3) **Depth of the model** :- Stacking multiple layers in the encoder also increases the depth of the model. Deeper models can learn more complex functions and capture more intricate relationships between input and output. This can lead to better performance on a variety of natural language processing tasks.

So the stack of  $N = 6$ , identical layers in the encoder of the Transformer architecture is needed to capture increasingly complex and long – range dependencies in the input sequence, make the model more robust to noise, and increase the depth of the model to learn more complex functions.

**What advantage will they give when the weights differ in those six layers ?**

Since each layer can perform different type of processing on the input, it allows the model to learn multiple representations of the same input.

Further more, each of these two sub layers has a residual connection around it.

**Question :- What advantage will this residual connection gives in the transformer architecture ?**

The residual connection is important component of the Transformer architecture for natural language processing tasks, including in the encoder part. The residual connection adds the original input to the output of each sub – layer in the Transformer layer, allowing information from the original input to flow through to the output of the layer. This has several advantages :

1) **Gradient propagation** :- The residual connection allows gradients to flow more easily through the network during training. Without the residual connection, gradients can diminish as they pass through multiple layer of the network, making it harder to train deep neural networks. With the residual connection, the gradients can flow more easily through the network, making it easy to train deeper models.



2) **Stabilization** :- The residual connection helps stabilize the learning process by preventing vanishing gradients. The residual connection allows information from the original input to be preserved in the output of each layer , preventing the gradients from becoming too small and vanishing. This can help prevent the model from getting stuck in local minima during training.

3) **Feature reuse** :- The residual connection allows the model to reuse features from earlier layers in the network. By adding the original input to the output of each layer , the model can reuse features from earlier layers that are still relevant from the current layer. This can help reduce the number of new features that need to be learned in each layer , which can be especially beneficial when working with limited amounts of data.

In summary , the residual connection in the encoder part of the Transformer architecture provides several advantages , including improved gradient propagation , stabilization of the learning process , and feature reuse. These advantages can help the Transformer architecture achieve state of the art performance on a wide range of natural language processing tasks.

**Question :-** How are we going to add the skip connection mathematically ?

Each sub layer is also succeeded by a normalization layer , layer norm (.) , which normalizes the sum of computed between the sub layer input  $x$  and the output generated by the sublayer( $x$ )

**layer norm ( $x + \text{sub layer}(x)$ )**

**What feature will this normalization add ?**

The transformer architecture for natural language processing task includes layer normalization in the encoder part , specifically in each sub – layer of the Transformer layer. Layer normalization has several advantages.

1) **Improved training stability** :- Layer normalization helps to stabilize the training process of the Transformer network by reducing the internal covariate shift. Covariate shift refers to the change in the distribution of the input to a layer , which can make the training process unstable. By normalizing the inputs to each layer , layer normalization reduces the internal covariate shift and stabilizes the training process.

2) **Improved generalization** :- Layer normalization can improve the generalization of the model by reducing overfitting. Overfitting occurs when a model learns to fit the training data too closely , resulting occurs when a model learns to fit the training data too closely , resulting in a poor performance on new , unseen data. Layer normalization can help prevent over fitting by reducing the impact of small changes in the input , which can lead to more robust and generalizable models.

3) **Improved Performance** :- Layer normalization has been shown to improve the performance of the Transformer network on various natural language processing tasks. By reducing the internal covariate shift and improving the generalization of the model , layer normalization can lead to better performance on tasks such as machine translation , language modeling and text classification.

In summary , layer normalization in the encoder part of the Transformer architecture provides several advantages , including improved training stability , improved generalization , and improved performance on natural language processing tasks. These advantages can help the Transformer architecture achieve state of the art performance on a wide range of natural language processing tasks.

**Question :-**

- ✓ What is Covariate shift deeply ?
- ✓ How does the normalization improve the generalization of the model and reduces over fitting ?
- ✓ How are we going to add the normalization layer to the output of the residual connection ?
- ✓ What is the mathematical process of normalization ?

Normalization can help prevent overfitting in neural networks by reducing the impact of small changes in the input. Overfitting occurs when a model becomes too complex and learns to fit the training data too closely , resulting in poor performance on new , unseen data. Normalization can help prevent overfitting by reducing the sensitivity of the model to small changes in the input , which can lead to more robust and generalizable models.

In neural networks , small changes in the input can have a large impact on the output of the network , especially in deep networks with many layers. Normalization can help reduce the impact of these small changes by scaling the input to each layer so that it has zero mean and unit variance. This can help reduce the sensitivity of the network to small changes in the input , making it more robust to variations in the training data.

By reducing the impact of small changes in the input , normalization can help prevent overfitting by encouraging the model to learn more generalizable features that are less sensitive to small variations in the training data. This can help the model perform better on new , unseen data , which is the ultimate goal of machine learning.

In summary , normalization helps prevent overfitting by reducing the sensitivity of the model to small changes in the input , which can lead to more robust and generalizable models.

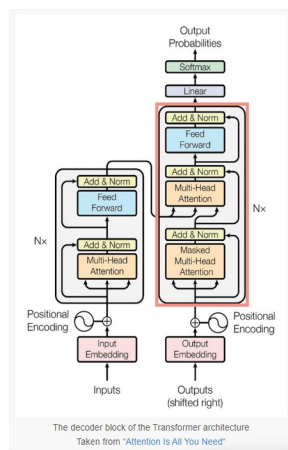
## POSITIONAL EMBEDDING

An important consideration to keep in mind is that the Transformer architecture can not inherently capture any information about the relative position of the words in the sequence since it does not make use of recurrence. This information has to be injected by introducing positional encoding to the input embeddings.

The positional encoding vectors are of the same dimension as the input embeddings and are generated using sine and cosine functions of different frequencies. Then, they are simply summed to the input embeddings in order to inject the positional information.

**Question :- How does the positional encoding works ? How does using sine and cosine functions used for generating the vectors of the positional embedding ?**

The Decoder



In languages, the order of the words and their position in a sentence really matters. The meaning of the entire sentence can change if the words are re-ordered. When implementing NLP solutions, recurrent neural networks have a built mechanism that deals with the order of sequences. The transformer model however does not use recurrence or convolution and treats each data point as independent of the other. Since the Transformer architecture ditched the recurrence mechanism in favor of multi-head self attention mechanism. Avoiding the RNN's methods of recurrence will result in massive speed up in the training time and theoretically, it can capture longer dependencies in a sentence.

## How does the Transformer architecture ditched the recurrence mechanism and maintain parallelism

The Transformer architecture, introduced in the paper "Attention is All You Need" by Vaswani et al(2017), replaces the recurrent neural network (RNN) mechanism typically used in sequence to sequence models with self attention mechanisms, which enables parallelization of computation.

In traditional RNN, the hidden state at each time depends on the hidden state of the previous time step, which

makes the computations sequential and not easily parallelizable. This can limit the performance of the model on longer sequences, since each time step must wait for the previous time step to compute.

In contrast, the Transformer architecture utilizes self attention mechanism, which allows the model to process all input tokens in parallel. Specifically, the model computes attention score for each token in the input sequence based on its relationship with every other token in the sequence. The attention scores are used to weight the importance of each token, which are then used to compute a weighted sum of all tokens in the sequence. This weighted sum, which captures the most relevant information from the input sequence, is then used as input to the subsequent layers of the model.

Since the attention mechanism consider all token in the input sequence in parallel, the Transformer architecture is more efficient and faster than traditional RNNs, and can process longer sequences without sacrificing performance. Additionally, the Transformer architecture has become a popular choice for various natural language processing tasks such as machine translation, text generation, and language understanding, due to its superior performance and parallelizability.

Here, positional information is added to the model explicitly to retain the information regarding the order of words in a sentence, positional encoding is the scheme through which the knowledge of the order of objects in a sequence is maintained.

## How does the positional encoding works ?

The Transformer architecture uses a positional encoding mechanism to inject position information in to the input sequence so that the self attention mechanism can differentiate between tokens based on their position in the sequence.

Since the self attention mechanism in the Transformer does not consider the order of the input sequence, it is important to incorporate the position of each token in to the input representation. The positional encoding achieves this by adding a vector to the embedding of each token that encodes the position in the sequence.

The positional encoding vector is fixed, sinusoidal function of the position, which is added to the input embeddings. Specifically, the positional encoding is calculated as follows :-

$$\begin{aligned} \text{pos\_enc}(\text{pos}, 2*i) &= \sin(\text{pos}/10000^{(2*i/d\_model)}) \\ \text{pos\_enc}(\text{pos}, 2*i+1) &= \cos(\text{pos}/10000^{(2*j/d\_model)}) \end{aligned}$$

Where "pos" is the position of the token in the sequence, "i" refers to position along embedding vector dimension and "d\_model" is the dimension of the embedding.

Each dimension of the positional encoding corresponds to different frequency of the sinusoidal function. By using different frequencies, the model can distinguish between tokens based on their position in the sequence.

The positional encoding is added to the input embedding at the beginning of the network and the resulting vectors are fed to the self attention layers. The self attention mechanism can then differentiate between tokens based on both their semantic meaning and their position in the sequence.

The positional encoding mechanism allows the Transformer to incorporate positional information into the input sequence without disrupting the parallelizability of the model.

**Why are we using a sinusoidal function for generating the positional encoding ?**

The Transformer architecture uses a sinusoidal function for generating positional encodings because it provides a fixed, continuous encoding for each position in the sequence that is easily learnable by the model.

A key requirement of the positional encoding function is that it must be fixed and known in advance, so that it can be added to the input embeddings before the model is trained. The use of a fixed positional encoding allows the model to generalize well to input sequences of different lengths and extrapolate to longer sequences during inference.

The choice of sinusoidal function is motivated by the fact that it is a continuous, periodic function that can represent any position in the sequence with a unique encoding. This ensures that each position in the sequence is mapped to distinct encoding, which is important for the self attention mechanism to distinguish between different positions.

Moreover, the choice of sinusoidal function with different frequencies ensures that each dimension of the encoding captures different aspects of the position information. Specifically, each dimension of the encoding corresponds to a different frequency of the sinusoidal function, allowing the model to capture positional information at different scales.

The use of sinusoidal function allows the positional encoding to be easily computed and added to the input embeddings using simple mathematical operations. This enables efficient computation and fast training of the Transformer model.

### The Formula of positional encoding

So what we are going to do is, we are going to embed some arbitrary word and then we are going to have a positional encoding for the arbitrary location and then we are going to add the two together and this is going to give us our encoding that we are going to send to our first layer of the encoder.

Positional encoding does not depend on the feature of any word, we are not looking at the word itself, we are only looking at the position of the word.

So our positional encoding formula uses sines and cosines and in the formula pos refers to the position of the word and "i" refers to the index of the d-dimensional hidden vector.

**Positional Encoding**

Authors propose using sinusoids to inject the information directly into a word embedding:

- $PE_{pos, 2i} = \sin\left(\frac{pos}{10000^{2i/d}}\right)$
- $PE_{pos, 2i+1} = \cos\left(\frac{pos}{10000^{2i/d}}\right)$

Position of the word: pos  
Index of d-dimensional vector: i

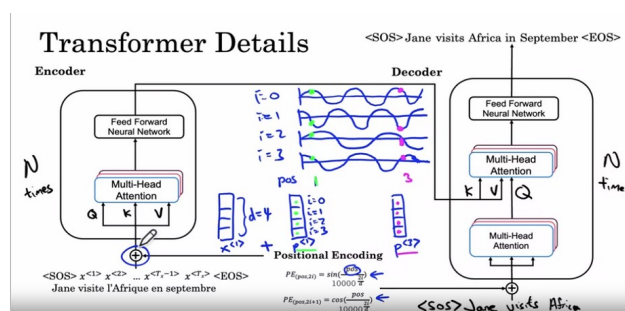
- Positional Encoding does not depend on the feature of any given word. Only the position that it appears in.
- In implementation, we have a PE matrix  $\in \mathbb{R}^{maxT \times d}$ .
- maxT is the maximum sequence length we ever want to support (e.g. 5000)
- d is our 'model dimensionality'

```

PE_matrix = zeros(maxT, d)
for pos in range(maxT):
    for i in range(d):
        if i % 2 == 0:
            PE_matrix[pos, i] = sin(pos/10000**(i/d))
        if i % 2 == 1:
            PE_matrix[pos, i] = cos(pos/10000**(i/d))
    
```

Then, given a word embedding and its position in the sequence (e.g. given: pos = 2, embedding = [1.0, 1.1, 1.2, ...]) add PE\_matrix[2] to the embedding

So in the first formula, 2i refers to the even index of the vector and in the second formula 2i + 1 refers to the odd index of the vector. So for all of the even indexes in the word one, we will use the first formula by making the position (pos) = 1 and for all of the odd indexes in the word one, we will use the second formula by making the position (pos) = 1



**Question :- why do we used sine for even indices and cos for odd indices ?**

In the positional encoding used in the transformer model in natural language processing, sine and cosine functions are used to generate different positional embeddings for each position in a sequence. Specifically, sine and cosine functions with different frequencies are used to encode the position of each token in the sequence.

The choice of using sine for even indices and cosine for odd indices is somewhat arbitrary, but it ensures that the positional embeddings generated for adjacent positions are different enough from each other to provide unique information to the model. If we used the same function for all positions, the model may not be able to distinguish between adjacent positions.

By using sine and cosine functions with different frequencies, we can ensure that the positional

embeddings for each position are unique and can be easily distinguished by the model.

#### Questions :-

- How does the variation of the frequency helps us to generate a unique vector

#### Why do we need the denominator to be 10000 ?

If it is small the frequency will be higher and the cyclic repetition would be faster( the period for being a cycle will be smaller ( faster ) since frequency and period are inversely proportional ). This increases the probability of positional encoding of different positions to be similar. Large omega increases the probability that different positions will have unique encodings.

- In the denominator of the positional encoding formula , what is the application of using  $(2i/d)$

- How does the positional encoding will be able to capture both fine – grained and coarse – grained positional information.

#### THE DECODER IN THE TRANSFORMER ARCHITECTURE ?

The decoder shares several similarities with the encoder. The decoder also consists of a stack of  $N = 6$  identical layers that are each composed of three sub layers.

1) The first sub layer receives the previous output of the decoder stack , augments it with positional information and implements multi – head self attention over it. While the encoder is designed to attend to all words in the input sequence regardless of their position in the sequence , the decoder is modified to attend only to the preceding words. Hence , the prediction for a word at position  $I$  can only depend on the known outputs for the words that come before it in the sequence.

#### Question :- Why do we need positional encoding for the decoder ?

We need positional encoding for the decoder in the Transformer architecture for the same reason we need it for the encoder , to provide the decoder with information about the relative positions of the input tokens.

In the Transformer architecture , the decoder generates an output sequence by attending to the encoded input sequence produced by the encoder. To attend to the input sequence correctly , the decoder needs to be aware of the relative positions of the input tokens , so that it can give appropriate weights to each token during the attention process.

Without positional encoding , the decoder would have no information about the relative positions of the input tokens , and would not be able to perform attention effectively. This could result in poor performance on

natural language processing tasks , such as language translation or text classification.

Therefore , like the encoder , the decoder in the Transformer architecture also uses positional encoding to encode the relative positions of the input tokens. The positional encoding is added to the decoder's input embeddings before they are fed in to the decoder layers , allowing the decoder to attend to the input sequence correctly and generate an appropriate output sequence.

#### Question :- What is the job of this masked multi – head attention specifically ?

#### Question :- Why are we masking the multi – head attention ?

The job of the masked multi-head attention mechanism in the decoder part of the Transformer architecture is to allow the decoder to attend to the previously generated tokens in the output sequence , while preventing it from attending to future tokens.

In the Transformer architecture , the decoder generates the output sequence one token at a time , in an auto-regressive manner. This means that at each step , the decoder generates a new token based on the previously generated tokens in the sequence. To generate each token , the decoder attends to the encoded input sequence produced by the encoder , as well as the previously generated tokens in the output sequence.

However , during training , we do not want the decoder to have access to future tokens in the output sequence , as this would result in data leakage and prevent the model from learning to generate outputs in a sequential and auto regressive manner. To prevent the decoder from attending to future tokens , the masked multi head attention mechanism is used.

The masked multi head attention mechanism works by masking out the attention scores for future tokens in the output sequence. Specifically , a binary mask is applied to the attention scores such that future tokens are assigned a score of negative infinity , which effectively prevents the decoder from attending to them. This ensures that the decoder can only attend to the previously generated tokens and the encoded input sequence when generating each new token in the output sequence.

Overall , the masked multi – head attention mechanism is an important component of the Transformer architecture that allows the decoder to generate output sequence in a sequential and auto regressive manner , while also preventing it from attending to future tokens during training.

2) The second layer implements a multi – head self attention mechanism similar to the one implemented in the first sub layer of the encoder. On the decoder side , this multi – head mechanism receives the queries from the previous decoder sub layer and the keys and the values from the output of the encoder. This allows the decoder to attend to all the words in the input sequence.



Question :- How does this work ? Why are we receiving the keys and queries from the encoder and the queries from the decoder.

3) The third layer implements a fully connected feed – forward network , similar to the one implemented in the second sub layer of the encoder.

Furthermore , the three sub-layer on the decoder side also have residual connections around them and are succeeded by a normalization layer.

Positional Encodings are also added to the input embeddings of the decoder in the same manner as previously explained for the encoder.

Question :- Why do we need a positional information for the decoder too ?

### How does the masked multi – head attention works

### MACHINE TRANSLATION EXAMPLE FOR UNDERSTANDING THE TRANSFORMER ARCHITECTURE

Starting again with the sentence “Jane Visite l’Afrique in Septembre ” and it’s corresponding embedding. Let’s walk through how we can translate the sentence from French to English. We’ve also added the start of sentence and end of sentence tokens here.

The first step in the transformer is these embeddings get fed in to an encoder block which has a multi head attention there. So this is exactly what we saw above where feed in the values Q , K and V computed from the embedding and the weight matrices W.

This layer then produces a matrix that can be passed in to feed forward neural network , which helps determine what interesting features there are in the sentence , in the transformer paper , the encoder block is repeated N times and a typical value for N is six. So after , may be about six times through this block , we will then feed this encoder in to a decoder block , and the decoder block’s job is to output the English translation.

So the first output will be the start of sentence token. At every step , the decoder block will input the first few words whatever we’ve already generated of the translation. When we’re just getting started , the only thing we know is that the translation will start with a start of sentence token and so the start sentence token gets fed in to this multi – head attention block.

And just this one token , the SOS token is used to compute Q , K , and V for this multi headed attention block. This first multi head attention output is used to generate the Q matrix for the next multi head attention block and the output of the encoder is used to generate K

and V. So the second multi headed attention block with inputs Q , K and V as before.

Question :- What does the SOS token holds , how does the start of token enabled us extract Q , K and V out of it ?

So whatever we have translated of the sentence so far will fed in to this masked multi head attention which will generate a query of the next word and pull context from K and V , which is translated from the french version of the sentence. To then try to decide what is the next word in the sequence to generate.

The multi head attention block outputs the values which are fed to feed forward neural networks. This decoder block is also going to be repeated N times , may be six times where we take the output , feed it back to the input and have this go through half a dozen times and the job of this neural network is to predict the next word in the sentence.

So hopefully It will decide that the first word in the English translation is Jane and what we do is then feed Jane to the input as well and now the next query comes from SOS and Jane and it says , well given Jane , what is the most appropriate next word. So we will find the right key and the right value that lets us generate the most appropriate next word , which hopefully will generate Visite and then running this neural network again generates Africa , then we feed Africa back in to the input. Hopefully it then generates en an then septembere and with this input hopefully it generates the end of sentence token and then we’re done.

These encoder and decoder blocks and how they are combined to perform a sequence translation tasks are the main ideas behind the transformer architecture.

If we recall the self attention equations , there is nothing that indicates the position of a word. So positional encoding to the input is very important to translation. The way we could encode the position elements in the input is that we use a combination of these sine and cosine equations.

So let’s say for example that our embedding is a vector with four values. In this case the dimension D of the word embedding is 4. So  $X_1$  ,  $X_2$  ,  $X_3$  let’s say those are four dimensional vectors. In this example we are going to then create a positional embedded in vector of the same dimension.

Let’s say for the position emebdding of the first word Jane (denoted as  $P_1$ ) and it denotes the numerical position of the word. Position Encoding did this with sine and cosine , which creates a unique position embedding vector for each word.

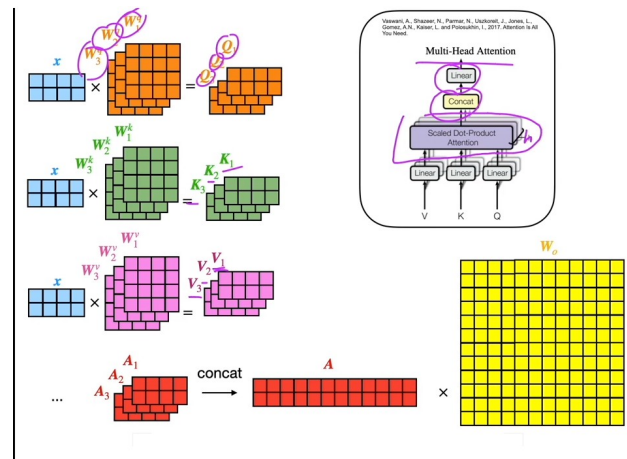
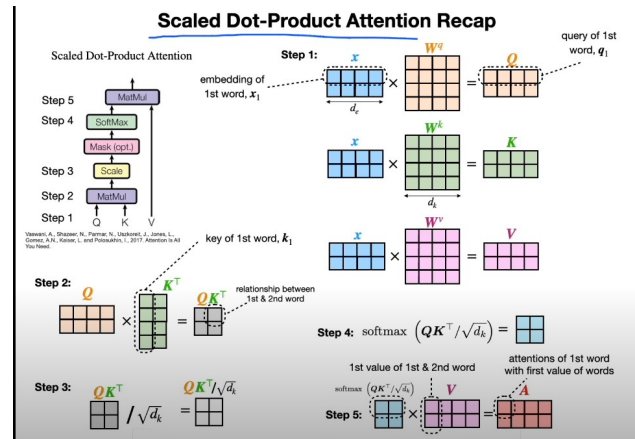
So this positional encodings will be added directly to the input embeddings so that each of the word vectors influence with where in the sentence the words appears.

The output of the encoding contains the contextual semantic embedding and positional encoding information.

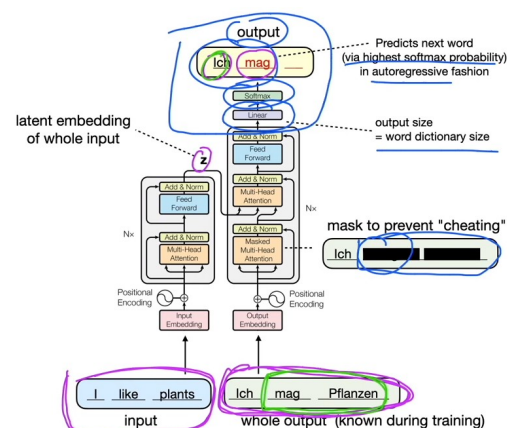
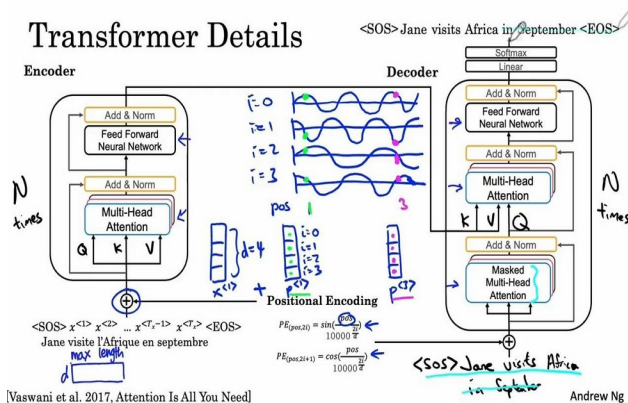
In addition to adding these positional encodings to the embedding, we would also pass them through the network with residual connections. These residual connections are similar to those we previously seen in the ResNet, and the purpose in this case is to pass along positional information through the entire architecture. In addition to positional information, the transformer network also uses a layer very similar to batch norm, for speeding up the learning process. And for the output of the decoder block, there is also a linear and then a softmax layer to predict the next word at a time.

In the literature of the transformer network, there is a masked multi head attention, which is important only during the training process where we are using a dataset of correct french to English translation to train our transformer. So previously we step through how the transformer prediction one word at the time, but how does it train? Let's say your dataset has the correct french to English Translation, Jane Visite l'Afrique en Septembre and Jane visits Africa in September. When training we have access to the entire correct English translation, the correct output and the correct input and because we have the full correct output we don't actually have to generate the words one at a time during training. Instead what masking does is it locks out the last part of the sentence to mimic what the network will need to do at test time during prediction. In other words, all that mask multi head attention does is repeatedly pretends that the network had perfectly translated say the first few words and hides the remaining words to see if given a perfect first part of the translation, whether the neural network can predict the next word in the sequence accurately.

## THE DIMENSION OF THE TRANSFORMER ARCHITECTURE



## Transformer Details



## Sum Up : The Transformer Model

The transformer model runs as follows :-

- 1) Each word forming an input sequence is transformed in to d-dimensional embedding vector.
- 2) Each embedding vector representing an input word is augmented summing it (element wise ) to a positional encoding vector of the same d-length hence introducing positional information in to the input.
- 3) The augmented embedding vectors are fed in to the encoder block consisting of the two sub layers explained above. Since the encoder attends to all words in the input sequence , irrespective if they precede or succeed the word under consideration , then the Transformer encoder is bi directional.
- 4) The encoder receives as input its own predicted output word at time step ,  $t-1$ .
- 5) The input to the decoder is also augmented by positional encoding in the same manner done on the encoder side.
- 6) The augmented decoder input is fed in to the three sub layers comprising the decoder block explained above. Masking is applied in the first sub layer in order to stop the decoder from attending to the succeeding words. At the second sub layer , the decoder also receives the output of the encoder , which now allows the decoder to attend to all the words in the input sequence.
- 7) The output of the decoder finally passed through a fully connected layer , followed by a softmax layer , to generate a prediction for the next word of the output sequence.

# Some Popular Transformer Models : BERT , GPT , BART

## BART TRANSFORMER AND DOWNSTREAM TASKS

BERT (Bidirectional Encoder Representation from Transformers ) is a transform-based language model architecture developed by Google. It is a pre-trained model that can be fine-tuned on a variety of NLP tasks , such as text classification , question answering , and named entity recognition.

The BERT architecture consists of several components , including :-

1) Input Embedding :- BERT takes as input a sequence of tokens , which are first converted in to embedding vectors using a combination of token embeddings and segment embeddings. The token embeddings represent the meaning of individual word , while the segment embeddings represent the context in which the word appear.

2) Transformer Encoder :- BERT uses a multi layer transformer encoder to process the input embeddings and generate contextualized representations of each token. The transformer encoder consists of multiple layers , each of which performs self-attention and feed-forward operations on the input embeddings.

3) Pre-training Objectives :- BERT is a pre-trained on two different objectives , masked language modeling (MLM) and next sentence prediction (NSP). MLM involves masking some of the input tokens and training the model to predict the original token from the context. NSP involves predicting whether two input sentences are consecutive or not.

4) Fine-Tuning Layers :- After pre-training , the BERT model can be fine-tuned on a variety of NLP tasks by adding task-specific layers on top of the pre-trained model. These layers are trained using task-specific labeled data and are used to make predictions for the specific task.

Overall , the BERT architecture is designed to generate a high-quality representation of natural language text that can be fine-tuned for a variety of NLP tasks.

### Question :-

- What does the name Bi-Directional Encoder Representations from Transformers represent ?
- Does the BERT architecture has a Decoder , if not why?
- What does the BERT architecture look like deeply ?

The name “Bidirectional Encoder Representations from Transformers ” (BERT) refers to the key features and components of the model architecture.

- Bidirectional :- BERT is a bidirectional model , meaning that it can process and understand text in both directions (from left to right and from right to left). This allows BERT to capture context and meaning from both the preceding and following words in a sentence.

- Encoder : BERT uses a multi-layer transformer encoder to process the input text and generate a contextualized representations of each token. The encoder is responsible for transforming the input text in to a series of hidden states that capture the meaning and context of each word.

- Representation :- BERT generates representations of the input text that can be used for a variety of downstream NLP tasks , such as text classification , question answering , and named entity recognition. These representations are based on the hidden states generated by the transformer encoder.

- Transformers :- BERT is based on the transformer architecture , which is a type of neural network that uses self attention to process input sequences. The transformer architecture is designed to capture long-range dependencies in sequences and is well suited for processing natural language text.

### Does the BERT architecture has a Decoder , if not why?

No , the BERT( Bidirectional Encoder Representation from Transformers ) architecture does not have a decoder. BERT is a type of transformer – based language model that is trained in an unsupervised manner using masked language modeling objective and next sentence prediction objective.

The transformer architecture used in BERT consists of an encoder that processes the input sequence and generates a sequence of hidden representations , which can be used for a variety of downstream natural language processing tasks such as text classification , named entity recognition and question answering.

In contrast to sequence-to-sequence models such as the Transformer and the GPT-series , which typically have both an encoder and a decoder , BERT is designed to perform tasks that do not require generating output sequences. Instead , BERT focuses on learning a deep bidirectional representation of natural language text that can be used to improve the performance of downstream tasks.

## Is it common to use BERT architecture for machine translation purposes ?

While BERT is a powerful architecture for natural language processing (NLP) tasks such as language modeling, text classification and question answering, it is commonly used for machine translation purposes.

BERT is a type of encoder-only transformer model, meaning it generates representations of text without a decoder component to produce an output sequence. In contrast, machine translation performance typically requires both an encoder and a decoder, such as in the popular seq2seq architecture.

However, BERT-based pre-training has been used as a feature extractor to improve machine translation performance. In this approach, the BERT model is fine-tuned on a large amount of monolingual data to learn sentence representations, and these representations are then used as inputs to machine translation system. The idea is that the rich contextual information captured by BERT can help improve the quality of the sentence representations used by the machine translation model.

Additionally, there are transformer-based models that are specifically designed for machine translation, such as the Transformer and its variants (example :- BART, T5). These models have both an encoder and decoder and are commonly used in the state of the art machine translation systems.

## Does question answering a seq2seq problem and needs an decoder based architecture than using encoder only architecture like BERT ?

Question answering is typically not considered as a Seq2Seq problem, but rather a reading comprehension problem. The goal of question answering is to find an answer to a question based on a given context, which can be a paragraph, a document, or a larger corpus.

While a Seq2Seq architecture with a decoder could be used for question answering, it is not the most common approach. Instead, the most popular approach for question answering is to use a pre-trained language model such as BERT or its variants.

In this approach, the context and the question are concatenated and fed as input to the BERT model. The model processes the input and generates a representation of the context and the question, which is used to predict the answer. The answer can be extracted using various techniques, such as identifying the answer span within the context or predicting the answer as a sequence.

Using a pre-trained language model such as BERT has shown to be highly effective for question answering tasks, and it is currently the state of the art approach for many such tasks.

## What does it mean by when we say "identifying the answer span within the context or predicting the answer as a sequence" ?

When we say "identifying the answer span within the context" or "Predicting the answer as a sequence" in the context of question answering, we are referring to two different approaches for extracting the answer from the input text.

In the first approach, we assume that the answer to the question is contained within the context and our goal is to identify the span of the text within the context that provides the answer. This is called extractive question answering and it involves finding the start and end positions of the answer span within the context.

For example :- given the context "The capital of France is Paris, and the population of Paris is around 2.2 million people" and the question "What is the capital of France?", the answer span is "Paris", and the start and end positions of the answer span can be identified by the model.

In the second approach, we treat the question answering task as a text generation task, where the goal is to generate the answer as a sequence of text. This is called generative question answering, and it involves predicting the answer as a sequence of words or tokens.

For example, given the same context and questions as before, the model might generate the answer as a sequence of text. This is called generative question answering, and it involves predicting the answer as a sequence of words or tokens.

For example, given the same context and question as before, the model might generate the answer sequence "The capital of France is Paris", without explicitly identifying the start and end positions of the answer span.

Both extractive and generative question answering approaches have their advantages and disadvantages, and the choice of approach depends on the specific task and requirements. Extractive question answering is often simpler and more interpretable, while generative question answering can handle more complex questions and situations where the answer is not explicitly stated in the context.

## BERT MODEL ARCHITECTURE

The BERT (Bidirectional Encoder Representations from Transformers) model architecture is a transformer-based neural network for natural language processing (NLP) tasks. It consists of a stack of transformer encoder layers that process the input sequence in a bi-directional manner, meaning that each token in the sequence is processed in both directions (from left to right and from right to left).

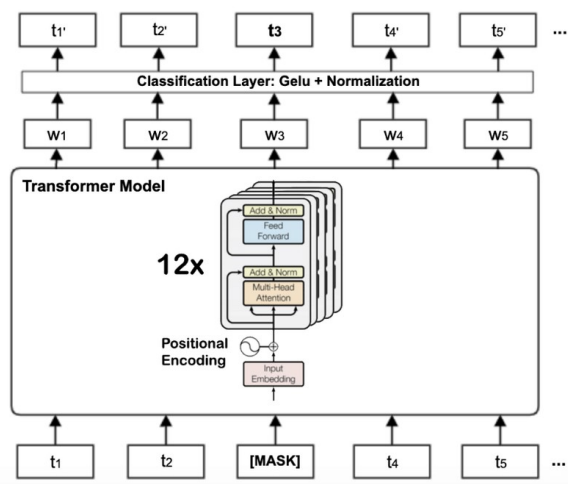
The input to the BERT model is a sequence of tokens, typically obtained by tokenizing a piece of text. The



tokens are first mapped to vector embeddings used an embedding layers , and then fed in to the stack of transformer encoder layers.

Each transformer encoder layer consists of two sub-layers : a self attention mechanism and a position – wise feed forward network. The self attention mechanism computes a weighted sum of the embeddings of all the tokens in the sequence , where the weights are computed based on the similarity between each token and all the other tokens in the sequence. The position wise feed forward network applies a non-linear transformation to each token embedding independently.

The output of the last transformer encoder layer is a sequence of hidden representations , where each hidden representation corresponds to one input token. This sequence of hidden representations can be used as input to downstream NLP tasks such as text classification , name entity recognition , and question answering.



The BERT model is trained in an unsupervised manner using two objectives : masked language modeling and next sentence prediction. Masked language modeling involves randomly masking some of the input tokens and predicting their original values based on the context of other tokens. Next sentence prediction involves predicting whether two given sentences are consecutive or not.

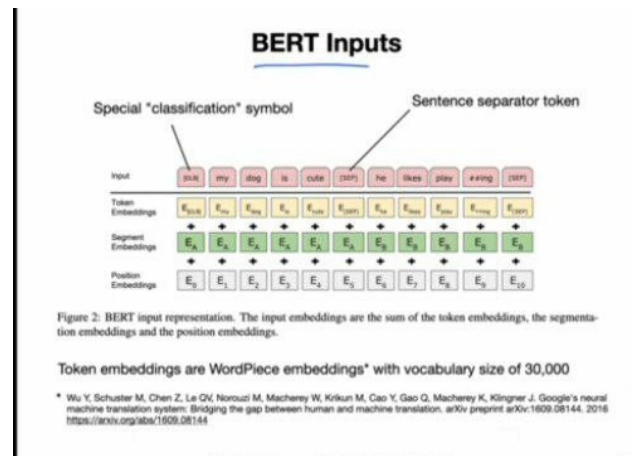
The pre-trained BERT model can be fine tuned on specific downstream NLP tasks by adding task-specific layers on top of the BERT model and training the entire model end to end on labeled dataset for the task.

**BERT INPUTS** :- The inputs to the BERT (Bidirectional Encoder Representations from Transformers) model architecture are word embeddings and segment embeddings.

Word embeddings are representations of words in a continuous vector space , which capture semantic and syntactic information about the words. In BERT , the word embeddings are obtained by concatenating a token

embedding and a positional embedding. The token embedding represents the meaning of the word , while the positional embedding encodes the position of the word in the input sequence.

Segment embeddings are used to distinguish between two different sentences or segments in the input. In BERT , the input sequence can contain two segments and each segments is assigned a different segment embedding. The segment embeddings are added to the token and positional embeddings to form the final input embeddings.



In addition to the input embeddings , BERT also uses various types of attention mechanism to capture the relationships between words in the input sequence. These include self-attention , which allows the model to attend to different parts of the input sequence when generating each output , and multi-head attention , which allows the model to attend to different aspects of the input sequences simultaneously.

### What are segment embeddings briefly :-

Segment embeddings are a type of input representation used in BERT (Bidirectional Encoder Representation from Transformers ) , a popular pre-trained language model for natural language processing (NLP) tasks.

In BERT , the model takes as input a sequence of tokens , which are individual words or sub words in the text , and produces a contextualized representation of each token based on its surrounding context. However , in some NLP tasks , such as question answering or natural language inference , it is important to distinguish between different segments of the input , such as the question and the answer or the premise and the hypothesis.

To address this , BERT introduces segment embeddings , which are additional input embeddings that indicate which segment each token belongs to. For example , consider the input sequence “The cat sat on the mat. It was fluffy.” Here there are two segments of text : the first segment is “The cat sat on the mat” and the second segment is “It was fluffy”.

In order for BERT to understand the relationship between segments, segment embeddings are used to indicate which tokens belong to which segment. In this case, the tokens in the first segment would be assigned a “segment A” embedding, while the tokens in the second segment would be assigned a “segment B” embedding. By using these segment embeddings, BERT can differentiate between the two segments and better understand the relationship between them, which is important for certain NLP tasks such as question answering and natural language inference.

Segment embeddings are added to the token embeddings and position embeddings which indicate the position of each token in the sequence. The resulting input embeddings are then fed in to the BERT model for further processing. By incorporating segment embeddings, BERT is able to better capture the relationships between different segments of the input and improve performance on tasks that require such understanding.

## BERT PRE-TRAINING TASKS

### BERT Pre-Training Tasks

#### Pre-training datasets

- BookCorpus (800 million words)
- Wikipedia (2500 million words)

#### Pre-training tasks

- Masked language model (“Cloze”)
- Next sentence prediction

Masked Language Modeling (MLM) is a key component of the pre-training phase of BERT (Bidirectional Encoder Representations from Transformers), a popular neural network architecture for natural language processing. (NLP)

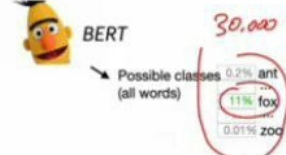
In MLM, some of the input tokens in a sequence (15 % of the words) are randomly replaced with a special token called “[MASK]”. The goal is for the BERT model to predict the original tokens based on their surrounding context, including both the tokens to the left and right of the masked token.

During pre-training, the BERT model is trained to minimize the difference between its predicted token distribution and the true token distribution. The true token distribution is obtained by comparing the predicted tokens against the original tokens in the sequence.

### BERT Pre-Training Task #1

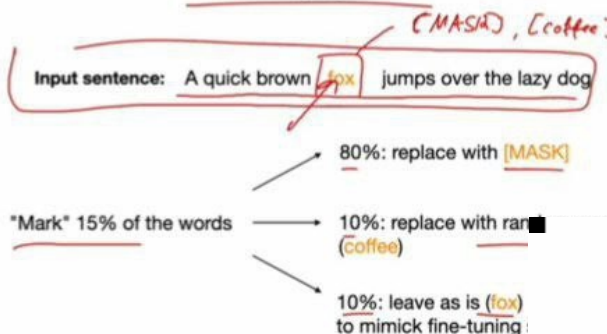
#### Masked Language Model

Input sentence: A quick brown fox jumps over the lazy dog  
Randomly masked: A quick brown [MASK] jumps over the lazy dog



### BERT Pre-Training Task #1

#### Masked Language Model



In addition to predicting the masked tokens, the BERT model is also trained to predict the next sentence in a text corpus. This is accomplished through a task called “Next Sentence Prediction (NSP)”, which involves providing two consecutive sentences as input to the BERT model and asking it to predict whether the second sentence follows the first sentence.

### BERT Pre-Training Task #2

#### Next Sentence Prediction

#### Balanced binary classification task (50% IsNext, 50% NotNext)

Input = [CLS] the man went to [MASK] store [SEP] he bought a gallon [MASK] milk [SEP]  
Label = IsNext

Input = [CLS] the man [MASK] to the store [SEP] penguin [MASK] are flight ##less birds [SEP]  
Label = NotNext

## BERT MASKED LANGUAGE MODEL

By combining MLM and NSP , BERT is able to capture wide range of language patterns and relationships. MLM helps the model to learn the relationships between words and phrases , while NSP helps it understand how different sentences relate to each other.

Overall , MLM allows BERT to learn a high quality representation of text that can be fine tuned on a variety of downstream NLP tasks , such as sentiment analysis , question answering and named entity recognition.

## Training BERT for downstream-tasks

### Transformer Training Approach

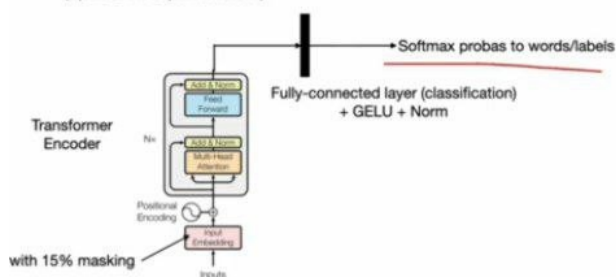
1. **Pre-training** on large unlabeled datasets (self-supervised learning)
2. **Training for downstream-tasks** on labeled data (supervised learning)
  - a) **fine-tuning** approach
  - b) **feature-based** approach (nowadays also called "fine-tuning")

Sebastian Raschka    STAT 453: Intro to Deep Learning

## BERT Pre-Training and Fine Tuning Approach

### BERT Pre-Training & Fine-Tuning Approach

- Add classification layer
- Train end-to-end on labeled dataset for downstream task (update ALL parameters)

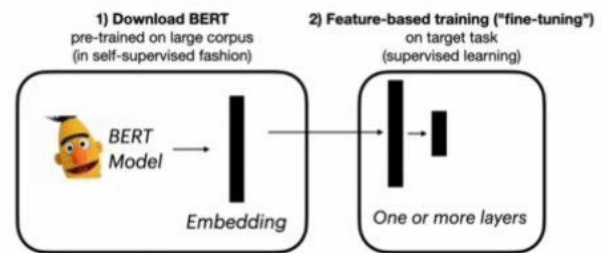


Sebastian Raschka    STAT 453: Intro to Deep Learning

## BERT PRE-TRAINING AND FEATURE BASED

### BERT Pre-Training & Feature-based Training

- Keep BERT frozen after pre-training
- Create BERT embeddings for labeled dataset for downstream task and train new model on these embeddings (in original paper, 2-layer biLSTM on embeddings from concatenated last 4 layers performed best)



Sebastian Raschka    STAT 453: Intro to Deep Learning

## GPT TRANSFORMER AND DOWN-STREAM TASKS

The GPT transformer architecture consists of a multi-layer bidirectional transformer encoder-decoder architecture , where the decoder is conditioned on the input text and is used to generate the output context. The encoder is used to extract features from the input text , which are then used by the decoder to generate the output text.

The GPT transformer architecture has several key components , including :-

1) **Embedding Layer** :- The input text is first transformed in to a sequence of embeddings , where each token is represented by a high-dimensional vector.

2) **Positional Encoding** :- The embedding vectors are augmented with a positional encoding vector that encodes the position of the token with in the sequence.

3) **Multi-head self attention** :- The transformer model uses multi-head self attention to attend to different parts of the input sequence. The attention mechanism allows the model to weigh the importance of each token in the sequence based on the relevance to the current token being processed.

4) **Feed Forward Network** :- The output of the attention layer is passed through a feed forward network , which applies non-linear transformations to the data.

5) **Layer Normalization** :- Each layer in the GPT transformer model has a layer normalization step , which normalizes the output of the feed forward network.

6) **Decoder** :- The decoder is conditioned on the input text and is used to generate the output text. The decoder consists of multiple layers of the same components as the encoder.

The GPT transformer model has been shown to be effective for a wide range of natural language processing

tasks , including language modeling , machine translation and text classification.

## How does the GPT Transformer differs from the ordinary transformer architecture ?

The GPT (Generative Pre-trained Transformer) transformer architecture is based on the original transformer architecture introduced in the paper “Attention is All You Need”. However , there are several key differences between the GPT transformer and the original transformer architecture.

1) Unidirectional Vs Bidirectional :- The original transformer architecture is bidirectional , which means that the decoder can attend to both past and future tokens in the input sequence. In contrast , the GPT transformer is unidirectional , which means that the decoder can only attend to past tokens in the input sequence. This makes the GPT transformer more suitable for generative tasks where the output is conditioned on the previous tokens in the input sequence.

2) Pre-training and fine-tuning :- The GPT transformer is typically pre-trained on a large corpus of text data using unsupervised learning techniques , such as language modeling. The pre-trained model can then be fine-tuned on a specific downstream tasks such as text classification or question answering. In contrast , the original transformer architecture is typically trained from scratch on a specific task.

3) Positional Encoding :- The GPT transformer uses a different method for positional encoding than the original transformer architecture. The GPT transformer uses learned positional embeddings that are added to the input embeddings , while the original transformer architecture uses fixed sinusoidal functions to encode the position of each token in the input sequence.

To account for the position of each token in the input sequence , the GPT transformer uses learned positional embeddings that are added to the input embeddings. These positional embeddings are learned during the training process and represent the position of each token in the sequence.

The learned positional embeddings are added to the input embeddings before they are processed by the multi-head self attention layer. This allows the model to capture the relative position of each token in the input sequence and attend to different parts of the sequence based on their position.

The use of learned positional embeddings is different from the original transformer architecture , which uses fixed sinusoidal functions to encode the position of each token in the input sequence. The use of learned positional embeddings in the GPT transformer allows the model to learn more flexible and task specific representations of the input sequence.

## UNDERSTANDING GPT FROM THE SCRATCH

GPT (Generative Pre-trained Transformer) is a deep learning model introduced by OpenAI in 2018. GPT is based on the Transformer architecture , which is a neural network architecture that is particularly suited for natural language processing (NLP) tasks such as language modeling , machine translation , and question answering.

The Transformer architecture consists of two main components : the encoder and the decoder. The encoder takes an input sequence of tokens and generates a sequence of encoded vectors , while the decoder takes the encoded vectors and generates a sequence of output tokens. GPT , in particular , is an auto-regressive language model based on the decoder-only Transformer architecture.

The GPT architecture consists of multiple layers of Transformer decoder blocks , each of which contains a self attention mechanism and a feed forward neural network. The self attention mechanism allows the model to attend to different parts of the input sequence and capture long – range dependencies , while the feed – forward neural network processes the attended inputs to generate the output sequence. The self attention mechanism works by computing a weighted sum of the input sequence tokens , where the weights are learned during training based on the similarity between the tokens.

The training process of GPT involves pre-training the model on a large corpus of text data in an unsupervised manner. During pre-training , the model learns to predict the next token in a sequence given the previous tokens. This is done by masking a random subset of the input tokens and training the model to predict the masked tokens. The pre-training is typically done using a variant of the stochastic gradient descent optimization algorithm , with a technique called back propagation used to adjust the model’s parameters to minimize the prediction error.

Once pre-trained , the GPT model can be fine tuned for specific NLP tasks such as text classification , language translation , and question answering. Fine – tuning involves training the model on a smaller labeled dataset specific to the task at hand , where the weights learned during pre-training are adjusted to better fit the task specific data. The fine tuning process typically involves freezing some layers of the model and updating the weights of the remaining layers using the labeled data.

One of the key strengths of GPT is its ability to generate high-quality text that is coherent and contextually relevant. This is achieved by leveraging the self-attention mechanism and the pre-training process , which allow the model to capture the relationships between different parts of the input sequence and generate outputs that are consistent with the input context.



In conclusion , GPT is a deep learning model based on the Transformer architecture that is particularly suited for natural language processing tasks. It is pre-trained on a large corpus of text data using an unsupervised learning approach and can be fine tuned for specific NLP tasks. GPT's strength lies in its ability to generate high quality text that is coherent and contextually relevant , making it a valuable tool for a wide range of NLP applications.

## THE GPT ARCHITECTURE

The GPT architecture consists of a stack of transformer decoder blocks , each of which contains a multi-head self attention mechanism , a position wise feed forward network and layer normalization. The self-attention mechanism allows the model to attend to different parts of the input sequence and capture long-range dependencies , while the feed forward network processes the attended input to generate the output sequence. Each decoder block takes the output of the previous block and generates a new output , and the final output is fed in to a softmax layer that generates the probability distribution over the output vocabulary.

## TRAINING GPT TRANSFORMER

The training of GPT is done in two phases : pre-training and Fine tuning. Pre-training is done on a large corpus data in an unsupervised manner , where the model is trained to predict the next token in a sequence given the previous tokens. This is done by masking a random subset of the input tokens and training the model to predict the masked tokens. The pre-training process is typically done using a variant of the stochastic gradient descent optimization algorithm , with a technique called back propagation used to adjust the model's parameters to minimize the prediction error.

Fine-tuning involves training the model on smaller labeled dataset specific to the task at hand , where the weights learned during pre-training are adjusted to better fit the task specific data. The fine tuning process typically involves freezing some layers of the model and updating the weights of the remaining layers using the labeled data.

## Downstream Tasks

GPT is a versatile architecture that is well suited for a wide range of downstream NLP tasks , including language modeling , text classification , sentiment analysis , question answering , and machine translation. Let us explore some of these tasks and how GPT helps for each of them.

1) Language Modeling :- GPT is particularly well suited for language modeling tasks , where the goal is to predict the next word in a sequence given the previous words. The architecture's self attention mechanism allows it to capture the relationships between different parts of the input sequence and generate outputs that are consistent with the input context. GPT has demonstrated state of the

art performance on language modeling tasks such as the PennTreebank and WikiText-103.

2) Text Classification :- GPT can also be used for text classification tasks , where the goal is to classify input text in to one or more predefined categories. For example , GPT has been used for sentiment analysis tasks , where the goal is to predict the sentiment of a piece of text (positive , negative or , neutral). The architecture's ability to capture the relationships between different parts of the input sequence and generate contextually relevant outputs makes it well suited for these tasks.

3) Sentiment Analysis :- GPT can also be used for sentiment analysis tasks , where the goal is to predict the sentiment of a piece of text (positive , negative , or neutral). The architecture's ability to capture the relationships between different parts of the input sequence and generate contextually relevant outputs makes it well suited for these tasks.

4) Question-Answering :- GPT can also be used for question-answering tasks , where the goal is to answer questions posed in natural language. For example , GPT has been used for the Stanford Question Answering Dataset (SQuAD) task , where the goal is to answer questions based on a given passage of text.

## FINE-TUNING GPT-TRANSFORMERS FOR GENERATIVE QUESTION ANSWERING

- ➔ Question Answering can be any of this
- ➔ QnA via Classification (The answer is categorical)
- ➔ QnA via Extraction (answer is In the text)
- ➔ QnA via the Language Modeling (Answer can be anything)

### Classification :-

If all our examples have Answer : X , where X is categorical (that is always "Good" , "Bad" etc ...) , you can do classification

In this set up , we would have text – label pairs.

### The Text Data :-

context :- Matt wrecked his car today ?  
Question :- How was Matt's day ?

Label :- Bad

For classification , we're probably better off fine tuning a BERT style model (something like RoBERTa).



**Extraction** :- If all our examples have Answer :- X , where X is a word (or consecutive words) , then it is probably best to do a SquAD-style fine tuning with a BERT-style model. In this setup , our inputs is text , start\_pos , end\_pos triplets.

**The Text Data** :-

Context :- In early 2012 , NFL commissioner roger Goodell stated that the league planned to make the 50<sup>th</sup> super Bowl “spectacular” and that it would be an important game for us as a league.

Question :- Who was the NFL commissioner in early 2012 ?

**Start Position and End position** :-

6 , 8

Note :- The start/end position values of course positions of tokens , so these values will depend on how we tokenize our inputs.

In this set up , we’re also better off using a BERT-style model.

**Language Modeling**

If all we our examples have Answer : X , where X can basically be anything (it need not be contained in the text , and is not categorical ) , then we would need to do language modeling.

In this set up , we have to use a GPT-style model , and our input would just be the whole text as is :

Context :- Matt wrecked his car today.

Question :- How was matt’s day ?

Answer : Bad

There is no need for label , since the text itself is the label (we are asking the model to predict the next word , for each word). Large models like GPT-3 should be good at these tasks without any fine tuning (if you give the right prompt and examples) but of course , these are accessed behind API’s. These platforms allow you to fine tune models (via language modeling ) .



















