ACADEMIC YEAR	2024 – 2025 ODD
YEAR	III
SEM	V
SUBJECT CODE	CSV14
SUBJECT TITLE	TEXT AND SPEECH ANALYSIS
FACULTY INCHARGE	P PRIYADHARSHINI

UNIT II TEXT CLASSIFICATION 9

Basics of Sentiment Analysis – Lexicon based approaches: Words and Vectors – Word2Vec Model – Glove Model – FastText Model. Deep Learning Architecture for Sequence Processing: RNN, Transformers – Text Classification Applications.

Sentiment analysis is a process that involves analyzing textual data such as social media posts, product reviews, customer feedback, news articles, or any other form of text to classify the sentiment expressed in the text. The sentiment can be classified into three categories: Positive Sentiment Expressions indicate a favorable opinion or satisfaction; Negative Sentiment Expressions indicate dissatisfaction, criticism, or negative views; and Neutral Sentiment Text expresses no particular sentiment or is unclear.

Before analyzing the text, some preprocessing steps usually need to be performed. These include tokenization, breaking the text into smaller units like words or phrases, removing stop words such as common words like “and,” “the,” and so on, and stemming or lemmatization, which involves reducing words to their base or root form. At a minimum, the data must be cleaned to ensure the tokens are usable and trustworthy.

The strategies vary in complexity as well. In order of complexity

- **Lexicon-Based Methods:** Using dictionaries or lists of terms and their associated sentiment scores to determine overall sentiment. Consider a list of terms closely associated with positive sentiment within a domain and map those terms to a body of text to decide a final classification.
- **Machine Learning and Deep Learning:** One approach to classify sentiments is to use supervised learning algorithms or neural networks. These methods rely on pre-labeled data to accurately categorize different emotions or opinions.
- **Hybrid Approaches:** Combining multiple methods to improve accuracy, like machine learning models and lexicon-based analysis.

Sentiment analysis has multiple applications, including understanding customer opinions, analyzing public sentiment, identifying trends, assessing financial news, and analyzing feedback.

Challenges in sentiment analysis

It can be challenging for computers to understand human language completely. They struggle with interpreting sarcasm, idiomatic expressions, and implied sentiments. Despite these challenges, sentiment analysis is continually progressing with more advanced algorithms and models that can better capture the complexities of human sentiment in written text.

VECTOR SEMANTICS AND EMBEDDINGS

Vector semantics is the standard way to represent word meaning in NLP, helping vector semantics us model many of the aspects of word meaning. The idea of vector semantics is to represent a word as a point in a multidimensional semantic space that is derived from the distributions of embeddings word neighbours. Vectors for representing words are called embedding's (although the term is sometimes more strictly applied only to dense vectors like word2vec) Vector Semantics defines semantics & interprets word meaning to explain features such as word similarity. Its central idea is: Two words are similar if they have similar word contexts.

In its current form, the vector model inspires its working from the linguistic and philosophical work of the 1950s. Vector semantics represents a word in multi-dimensional vector space. Vector model is also called Embeddings, due to the fact that a word is embedded in a particular vector space. The vector model offers many advantages in NLP. For example, in sentimental analysis, sets up a boundary class and predicts if the sentiment is positive or negative (a binomial classification). Another key practical advantage vector semantics i of s that it can learn automatically from text without complex labelling or supervision. As a result of these advantages, vector semantics has become a de-facto standard for NLP applications such as Sentiment Analysis, Named Entity Recognition (NER), topic modelling, and so on.

WORD EMBEDDINGS It is an approach for representing words and documents. Word Embedding or Word Vector is a numeric vector input that represents a word in a lower-dimensional space. It allows words with similar meaning to have a similar representation. They can also approximate meaning. A word vector with 50 values can represent 50 unique features. Features: Anything that relates words to one another. Eg: Age, Sports, Fitness, Employed etc. Each word vector has values corresponding to these features.

Goal of Word Embeddings

- To reduce dimensionality
- To use a word to predict the words around it
- Inter word semantics must be captured

How are Word Embeddings used?

- They are used as input to machine learning models.
- Take the words —-> Give their numeric representation —-> Use in training or inference
- To represent or visualize any underlying patterns of usage in the corpus that was used to train them.

Implementations of Word Embeddings:

Word Embeddings are a method of extracting features out of text so that we can input those features into a machine learning model to work with text data. They try to preserve syntactical and semantic information. The methods such as Bag of Words(BOW), Count Vectorizer and TFIDF rely on the word count in a sentence but do not save any syntactical or semantic information. In these algorithms, the size of the vector is the number of elements in the vocabulary. We can get a sparse matrix if most of the elements are zero. Large input vectors will mean a huge number of weights which will result in high computation required for training. Word Embeddings give a solution to these problems. Let's take an example to understand how word vector is generated by taking emoticons which are most frequently used in certain conditions and transform each emoji into a vector and the conditions will be our features.

Happy	????	????	????
Sad	????	????	????

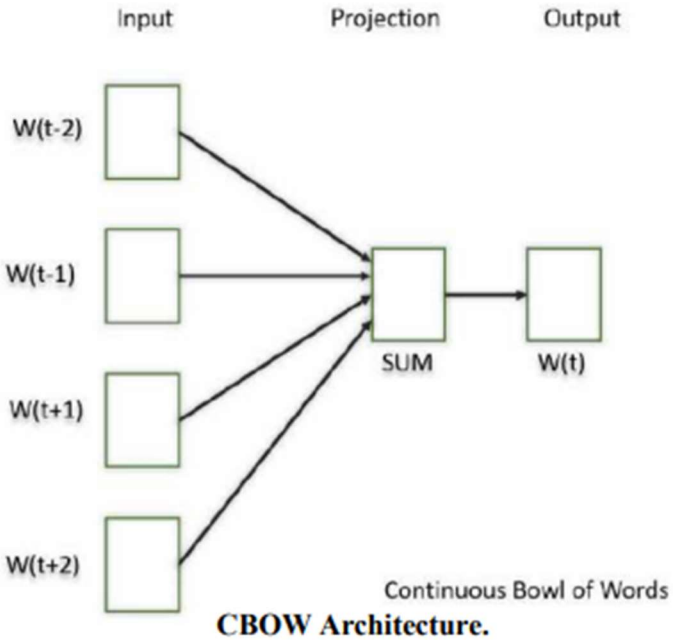
The emoji vectors for the emojis will be:

[happy, sad, excited, sick]
???? = [1,0,1,0]
???? = [0,1,0,1]
???? = [0,0,1,1] .....

In a similar way, we can create word vectors for different words as well on the basis of given features. The words with similar vectors are most likely to have the same meaning or are used to convey the same sentiment.

There are two different approaches for getting Word Embeddings:

1) Word2Vec: In Word2Vec every word is assigned a vector. We start with either a random vector or one-hot vector. One-Hot vector: A representation where only one bit in a vector is 1. If there are 500 words in the corpus then the vector length will be 500. After assigning vectors to each word we take a window size and iterate through the entire corpus. While we do this there are two neural embedding methods which are used: 1.1) Continuous Bowl of Words (CBOW) In this model what we do is we try to fit the neighbouring words in the window to the central word.

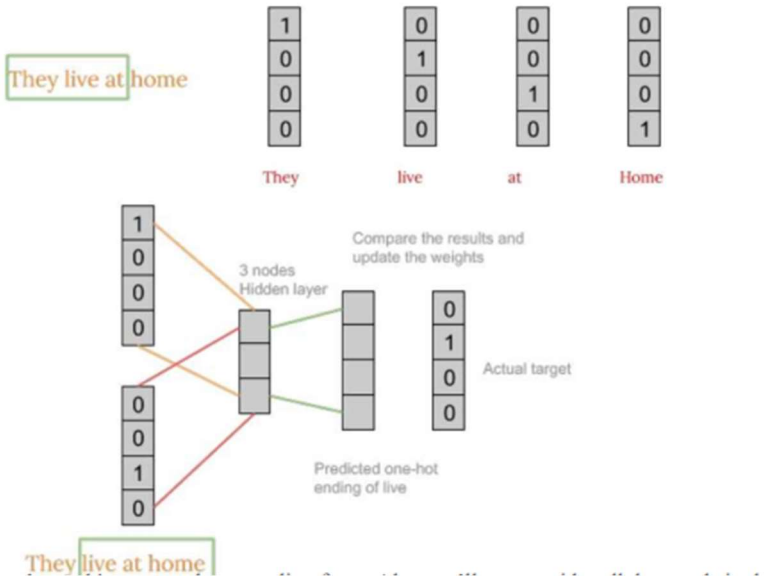


CBOW Architecture.

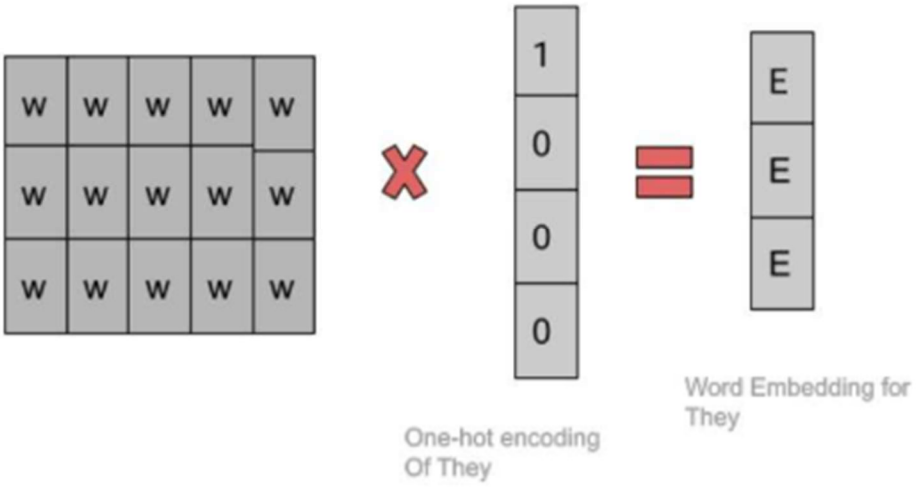
This architecture is very similar to a feed-forward neural network. This model architecture essentially tries to predict a target word from a list of context words.

The intuition behind this model is quite simple: given a phrase "Have a great day", we will choose our target word to be "a" and our context words to be ["have", "great", "day"]. What this model will do is take the distributed representations of the context words to try and predict the target word.

Consider a small example in which we have only four words i.e. live, home, they and at. For simplicity, we will consider that the corpus contains only one sentence, that being, 'They live at home'.



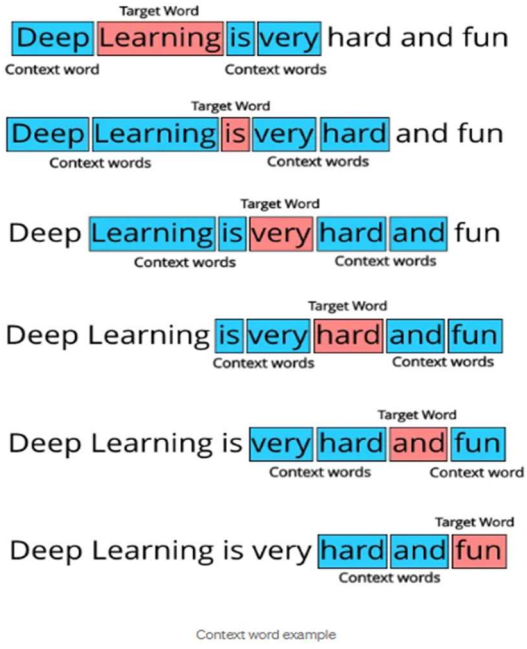
First, we convert each word into a one-hot encoding form. Also, we'll not consider all the words in the sentence but ll only take certain words that are in a window. For example, for a window size equal to three, we only consider three words in a sentence. The middle word is to be predicted and the surrounding two words are fed into the neural network as context. The window is then slid and the process is repeated again.
Finally, after training the network repeatedly by sliding the window a shown above, we get weights which we use to get the embeddings as shown below.



In this model, we try to make the central word closer to the neighbouring words. It is the complete opposite of the CBOW model. It is shown that this method produces more meaningful embeddings.

Example – 2

Consider, "Deep Learning is very hard and fun". We need to set something known as ***window size***. Let's say 2 in this case. What we do is iterate over all the words in the given data, which in this case is just one sentence, and **then consider a window of word which surrounds it**. Here since our window size is 2 we will consider 2 words behind the word and 2 words after the word, hence each word will get 4 words associated with it. We will do this for each and every word in the data and collect the word pairs.



Context word example

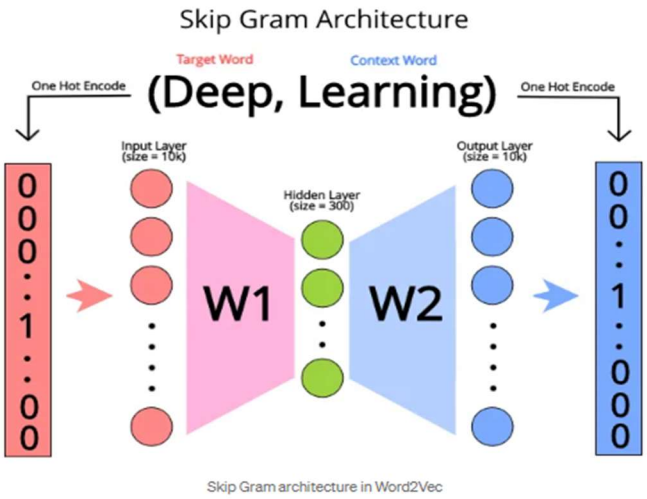
As we are ***passing the context window*** through the text data, we find all **pairs of target and context** words to form a dataset in the format of target word and context word. For the sentence above, it will look like this:
1st Window pairs: (Deep, Learning), (Deep, is)
2nd Window pairs: (Learning, Deep), (Learning, is), (Learning, very)
3rd Window pairs: (is, Deep), (is, Learning), (is, very), (is, hard)

And so on. At the end our **target word vs context word data set** is going to look like this:
(Deep, Learning), (Deep, is), (Learning, Deep), (Learning, is), (Learning, very), (is, Deep), (is, Learning), (is, very), (is, hard), (very, learning), (very, is), (very, hard), (very, and), (hard, is), (hard, very), (hard, and), (hard, fun), (and, very), (and, hard), (and, fun), (fun, hard), (fun, and)

This can be considered as our ***"training data"*** for word2vec.

In skipgram model, we try to predict each context word given a target word.

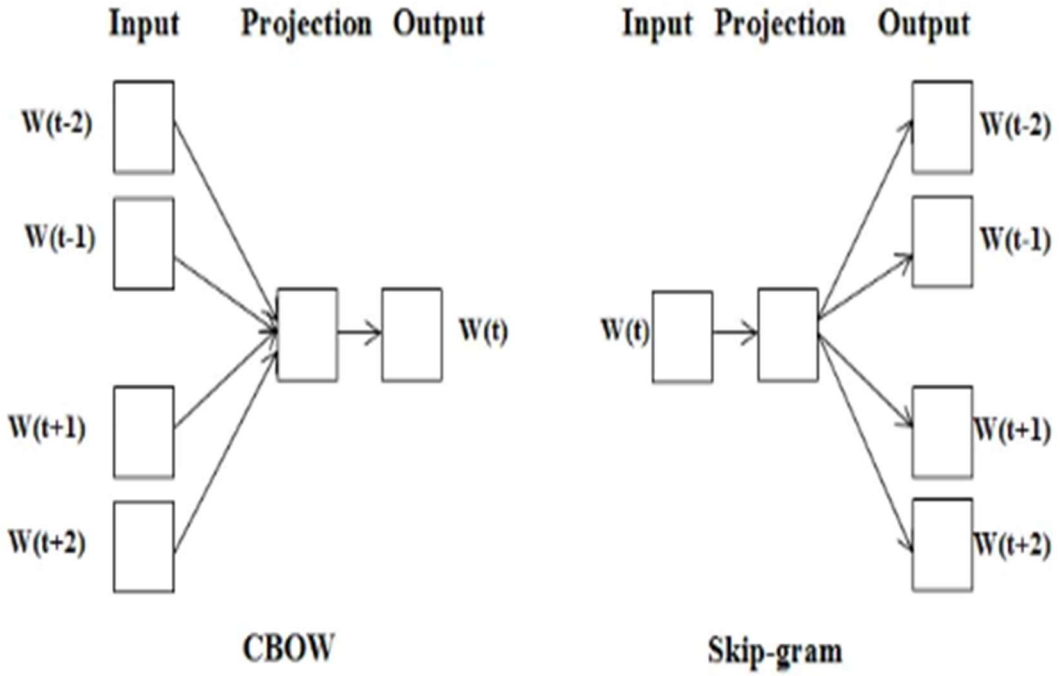
We use neural network for this prediction task. **The input to the neural network is the one hot encoded version of the context word**. Hence the size of the input and output layer is **V**(vocabulary count). This neural network has only one layer in the middle, the **size of the hidden layer determines the size of the word vectors we wish to have at the end.**



Since this neural network has a total of 3 layers, there will be only 2 weight matrices for the network, **W1** and **W2**. **W1** will have dimensions of 10000*300 and **W2** will have dimensions of 300*10000. These two weight matrices will play an important role in calculating word vectors.

For the entire dataset which we have collected from the original textual data, we will pass each pair into the neural network and train it. **Neural network here is trying to guess which context words can appear given a target word.** After training the neural network, if we input any target word into the neural network, it will give a vector output which represents the words which have a high probability of appearing near the given word.

For CBOW the only difference is that we try to predict the target word given the context words, essentially we just invert the skip gram model to get the CBOW model. It looks like this:



Here when we give a vector representation of a **group of context words**, we will get the **most appropriate target word which will be within the vicinity of those words**.

For example, if we give the sentence: Deep _____ is very hard, where ["Deep", "is", "very", "hard"] represents the context words, the neural network should hopefully give "Learning" as the **output target word**. This is the core task the neural network tries to train for in the case of CBOW.

Word vectors help represent semantics of the words — What does this mean?
It we means we could use vector reasoning for words one of the most famous example is from **Mikolov's paper**, where we see that if we use the word vectors and perform (here, we use **V(word)** to represent the vector representation of the word) V(King) -V(Man) + V(Woman), and the resulting vector is closest to V(Queen). It is easy to see why this is remarkable — our intuitive understanding of these words is reflected in the learned vector representations of the words.

This gives us the ability to add more of a punch in our text analysis pipelines-having an intuitive semantic representation of vectors will come in handy more than once.

Word2Vec takes as its input a large corpus of text and produces a vector space, typically of several hundred dimensions, with each unique words in the corpus being assigned a corresponding vector in the space. Word vectors are positioned to vector space such that words that share common contexts in the corpus are located in close proximity to one other in the space.

Glove
Glove is based on **matrix factorization technique on word context matrix**. It first **constructs** a **large matrix** of (words x context) co-occurrence information ie. for each word, you count how frequently we see those word in some context in a large corpus.
In order to understand how GloVe works, we need to understand 2 main methods which GloVe was built on –

1. Global matrix factorization: In NLP, global matrix factorization is the process of using matrix factorization form linear algebra to reduce large term frequency matrices. These matrices usually represent the occurrences or the absence of words in the document.
2. Local context window: Local context window methods are CBOW and Skip-Gram, the one which were explained above.

Glove is a word vector representation method where training is performed on aggregated global word-word co-occurrence statistics from the corpus. This means that like word2vec it uses context to understand and create the word representations. The research paper describing the method is called *GloVe: Global Vectors for Word Representation* and is well worth a read as it describes some of the drawbacks of LSA and Word2Vec before describing their own method.

The author of the paper mention that instead of learning the raw occurrence probabilities, it was more useful to learn ratios of these co-occurence probabilities. The embeddings are optimized, so that the dot product of 2 vectors equals the log of number of times the 2 words will occur near each other.
For example, if 2 words "cat" and "dog" occur in the context of each other, say 20 times in 10-word window in the document corpus, then —
Vector(cat).Vector(dog) = log(10)
This forces the model to encode the frequency distribution of words that occur near them in a more global context.

FastText

FastText is a vector representation technique developed by facebook AI research. As its name suggests its fast and efficient method to perform same task and because of the nature of its training method, it ends up learning morphological details as well.

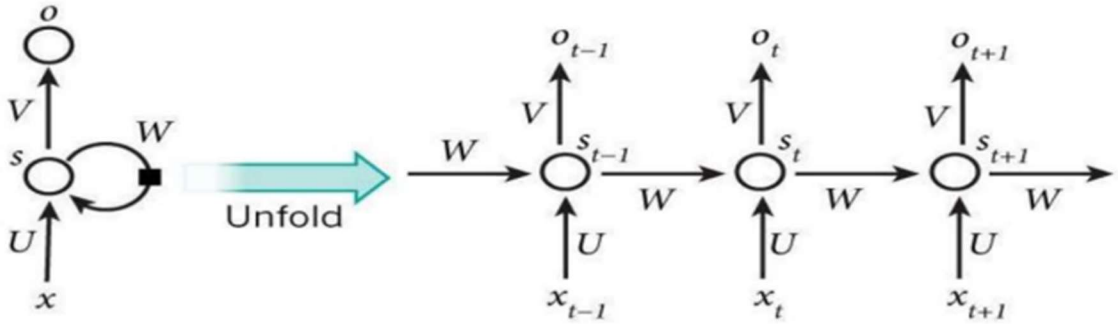
FastText is unique because it can derive word vectors for unknown words or out of vocabulary words — this is because by taking morphological characteristics of words into account, it can *create* the word vector for an unknown word. Since morphology refers to the structure or syntax of the words, FastText tends to perform better for such task, word2vec perform better for semantic task.
FastText works well with rare words. So even if a word wasn't seen during training, it can be broken down into n-grams to get its embeddings.
Word2vec and GloVe both fail to provide any vector representation for words that are not in the model dictionary. This is a huge advantage of this method.

Applications

- Analysing survey responses .
- Analysing verbatim comments.
- Music/Video recommendation system.

RNN - Recurrent Neural Network (RNN)

Recurrent Neural Network (RNN) is a type of Neural N etwork where the output from the previous step is fed as input to the current step. In traditional neural networks, all the inputs and outputs are independent of each other, but in cases when it is required to predict the next word of a sentence, the previous words are required and hence there is a need to remember the previous words. Thus, RNN came into existence, which solved this issue with the help of a Hidden Layer. The main and most important feature of RNN is its Hidden state, which remembers some information about a sequence. The state is also referred to as Memory State since it remembers the previous input to the network. It uses the same parameters for each input as it performs the same task on all the inputs or hidden layers to produce the output. This reduces the complexity of parameters, unlike other neural networks.



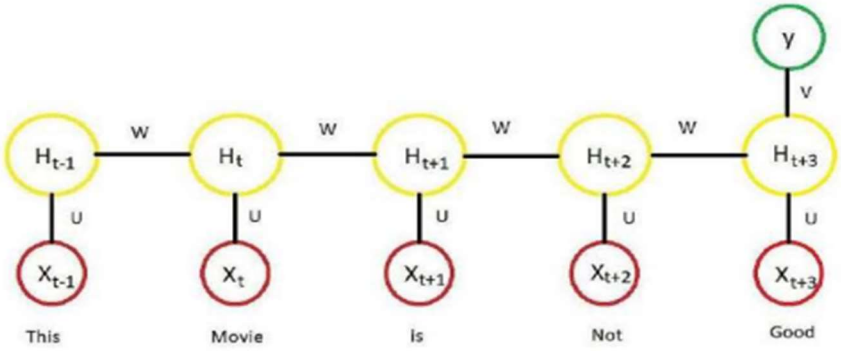
Recurrent neural network

The diagram above shows a detailed structure of an RNN architecture. The architecture described above is also called as a many to many architectures with (Tx = Ty) i.e. number of inputs = number of outputs. Such structure is quite useful in Sequence modelling.

Apart from the architecture mentioned above there are three other types of architectures of RNN which are commonly used.

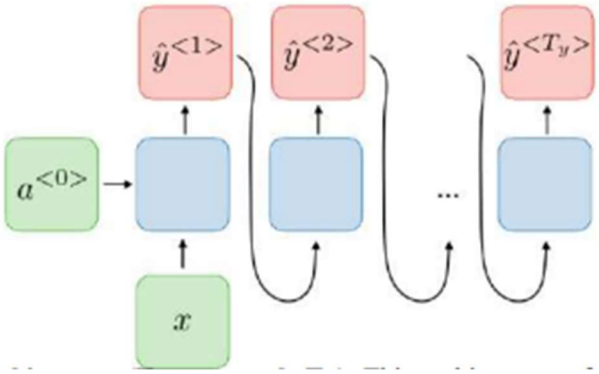
1.Many to One RNN: Many to one architecture refers to an RNN architecture where many inputs (Tx) are used to give one output (Ty). A suitable example for using such an architecture will be a classification task.

RNN are a very important variant of neural networks heavily used in Natural Language Processing. Conceptually they differ from a standard neural network as the standard input in a RNN is a word instead of the entire sample as in the case of a standard neural network. This gives the flexibility for the network to work with varying lengths of sentences, something which cannot be achieved in a standard neural network due to its fixed structure. It also provides an additional advantage of sharing features learned across different positions of text which cannot be obtained in a standard neural network.

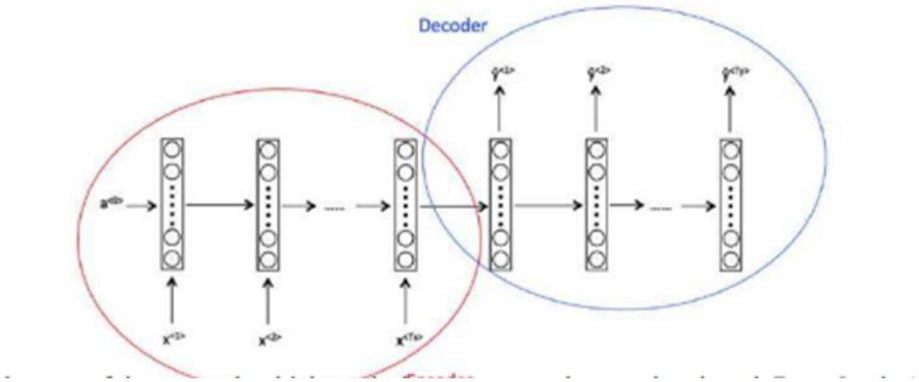


In the image above H represents the output of the activation function.

2. One to Many RNN: One to Many architecture refers to a situation where a RNN generates a series of output values based on a single input value. A prime example for using such an architecture will be a task, where a music generation input is a jounre or the first note.



3. Many to Many Architecture (Tx not equals Ty): This architecture refers to where many inputs are read to produce many outputs, where the length of inputs is not equal to the length of outputs. A prime example for using such an architecture is machine translation tasks.



Encoder refers to the part of the network which reads the sentence to be translated, and, is the part of the Decoder network which translates the sentence into desired language.

Limitations of RNN

Apart from all of its usefulness RNN does have certain limitations major of which are:

1. Examples of RNN architecture stated above can capture the dependencies in only one direction of the language. Basically, in the case of Natural Language Processing it assumes that the word coming after has no effect on the meaning of the word coming before. With our experience of languages, we know that it is certainly not true.
2. RNN are also not very good in capturing long term dependencies and the problem of vanishing gradients resurface in RNN.

NNs are ideal for solving problems where the sequence is more important than the individual items themselves. An RNNs is essentially a fully connected neural network that contains a refactoring of some of its layers into a loop. That loop is typically an iteration over the addition or concatenation of two inputs, a matrix multiplication and a non-linear function. Among the text usages, the following tasks are among those RNNs perform well at:

- Sequence labelling
- Natural Language Processing (NLP) text classification
- Natural Language Processing (NLP) text generation

Other tasks that RNNs are effective at solving are time series predictions or other sequence predictions that aren't image or tabular-based. There have been several highlighted and controversial reports in the media over the advances in text generation, Open AI's GPT-2 algorithm. In many cases the generated text is often indistinguishable from text written by humans.

RNNs effectively have an internal memory that allows the previous inputs to affect the subsequent predictions. It's much easier to predict the next word in a sentence with more accuracy, if you know what the previous words were. Often with tasks well suited to RNNs, the sequence of the items is as or more important than the previous item in the sequence.

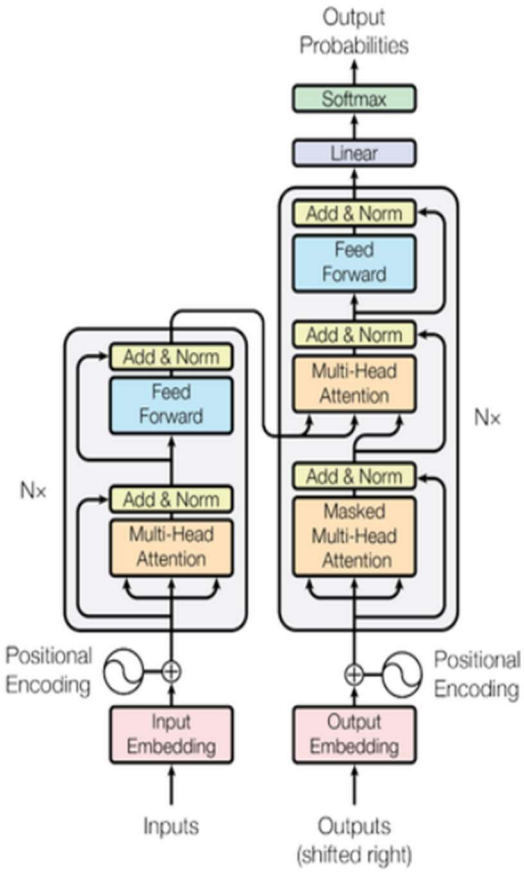
Transformer

"The Transformer is the first transduction model relying entirely on self-attention to compute representations of its input and output without using sequence-aligned RNNs or convolution."

Here, "transduction" means the conversion of input sequences into output sequences. The idea behind Transformer is to handle the dependencies between input and output with attention and recurrence completely.

Let's take a look at the architecture of the Transformer below. It might look intimidating but don't worry, we will break it down and understand it block by block.

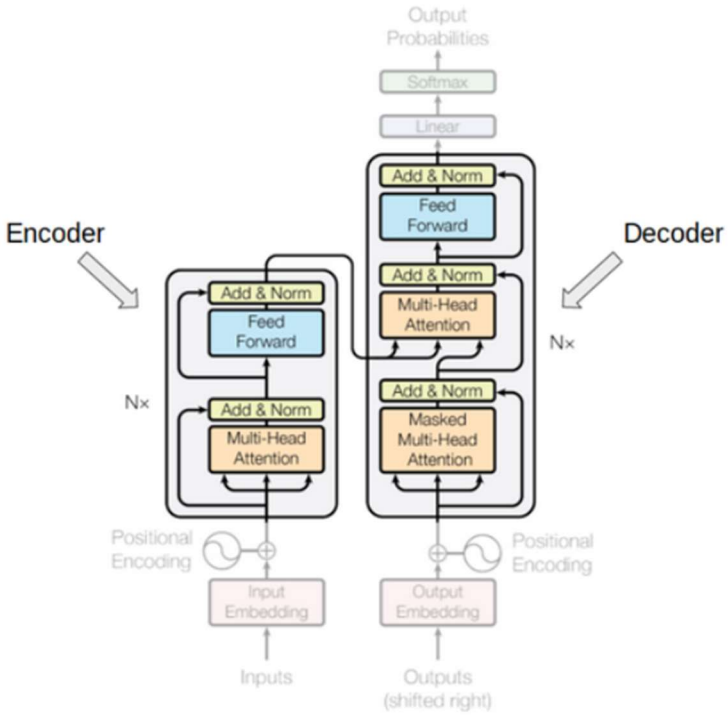
Transformer's Model Architecture



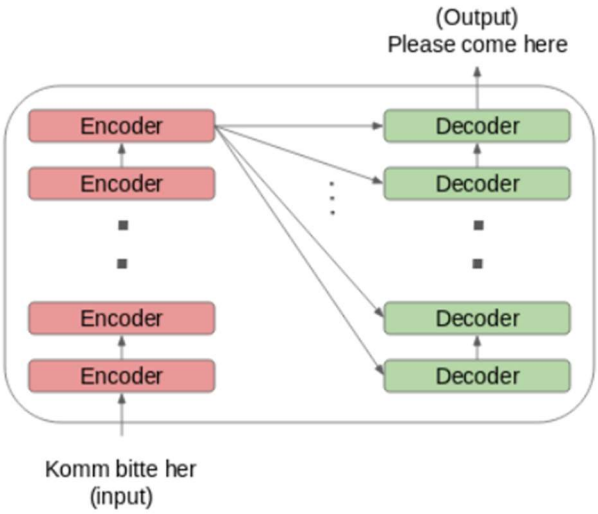
The Transformer – Model Architecture

The above image is a superb illustration of transformer in NLP architecture. Let's first focus on the Encoder and Decoder parts only.

Now focus on the below image. The Encoder block has 1 layer of a Multi-Head Attention followed by another layer of Feed Forward Neural Network. The decoder, on the other hand, has an extra Masked Multi-Head Attention.

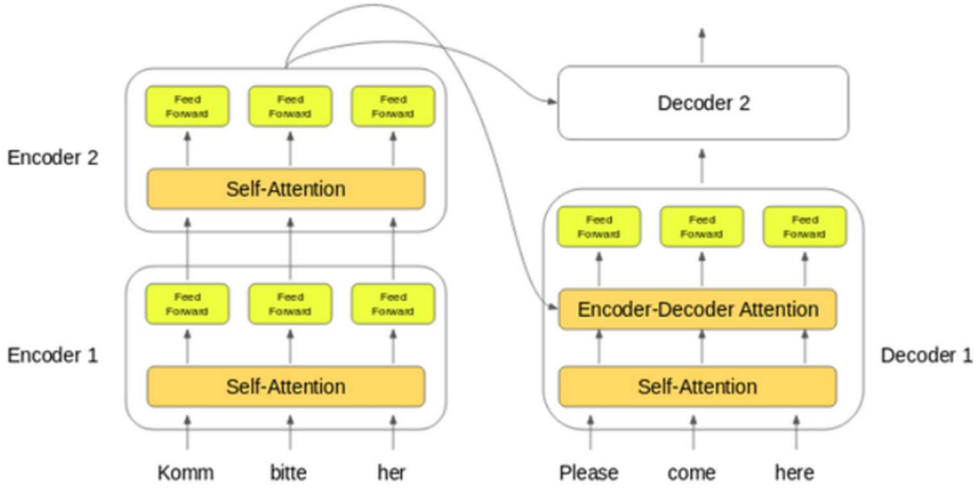


The encoder and decoder blocks are actually multiple identical encoders and decoders stacked on top of each other. Both the encoder stack and the decoder stack have the same number of units.The number of encoder and decoder units is a hyperparameter.



Let's see how this setup of the encoder and the decoder stack works:

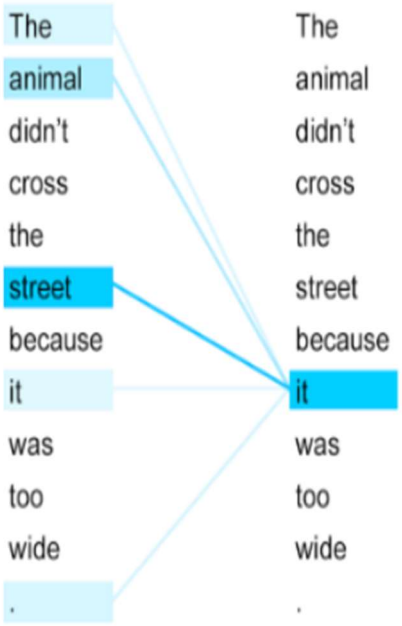
- The first encoder receives the word embeddings of the input sequence.
- It then transforms and propagates them to the next encoder.
- The output from the last encoder in the encoder-stack is passed to all the decoders in the decoder-stack, as depicted in the figure below:



An important thing to note here – in addition to the **self-attention** and feed-forward layers, the decoders also have one more layer of Encoder-Decoder Attention layer. This helps the decoder focus on the appropriate parts of the input sequence.

Getting a Hang of Self-Attention

"Self-attention, sometimes called intra-attention, is an attention mechanism relating different positions of a single sequence in order to compute a representation of the sequence."



Take a look at the above image. Can you figure out what the term **"it"** in this sentence refers to?

Is it referring to the street or to the animal? It's a simple question for us but not for an algorithm. When the model is processing the word "it", self-attention tries to associate "it" with **"animal"** in the same sentence.

Self-attention allows the model to look at the other words in the input sequence to get a better understanding of a certain word in the sequence. Now, let's see how we can calculate self-attention.

Calculating Self-Attention

First, we need to create three vectors from each of the encoder's input vectors:

- Query Vector
- Key Vector
- Value Vector.

During the training process, we train and update these vectors. We'll gain a deeper understanding of their roles once we complete this section.

Next, we will calculate self-attention for every word in the input sequence

Consider this phrase – "Action gets results". To calculate the self-attention for the first word "Action", we will calculate scores for all the words in the phrase with respect to "Action". This score determines the importance of other words when we are encoding a certain word in an input sequence

1. Taking the dot product of the Query vector (q1) with the keys vectors (k1, k2, k3) of all the words calculates the score for the first word:

Word	q vector	k vector	v vector	score
Action	q_1	k_1	v_1	$q_1 . k_1$
gets		k_2	v_2	$q_1 . k_2$
results		k_3	v_3	$q_1 . k_3$

2. Then, these scores are divided by 8 which is the square root of the dimension of the key vector:

Word	q vector	k vector	v vector	score	score / 8
Action	q_1	k_1	v_1	$q_1 . k_1$	$q_1 . k_1 / 8$
gets		k_2	v_2	$q_1 . k_2$	$q_1 . k_2 / 8$
results		k_3	v_3	$q_1 . k_3$	$q_1 . k_3 / 8$

3. Next, these scores are normalized using the softmax activation function:

Word	q vector	k vector	v vector	score	score / 8	Softmax
Action	q_1	k_1	v_1	$q_1 . k_1$	$q_1 . k_1 / 8$	x_{11}
gets		k_2	v_2	$q_1 . k_2$	$q_1 . k_2 / 8$	x_{12}
results		k_3	v_3	$q_1 . k_3$	$q_1 . k_3 / 8$	x_{13}

4. These normalized scores are then multiplied by the value vectors (v1, v2, v3) and sum up the resultant vectors to arrive at the final vector (z1).

This is the output of the self-attention layer. It is then passed on to the feed-forward network as input.

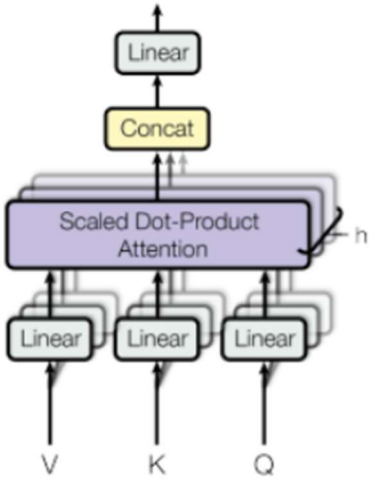
Word	q vector	k vector	v vector	score	score / 8	Softmax	Softmax * v	Sum
Action	q_1	k_1	v_1	$q_1 . k_1$	$q_1 . k_1 / 8$	x_{11}	$x_{11} * v_1$	z_1
gets		k_2	v_2	$q_1 . k_2$	$q_1 . k_2 / 8$	x_{12}	$x_{12} * v_2$	
results		k_3	v_3	$q_1 . k_3$	$q_1 . k_3 / 8$	x_{13}	$x_{13} * v_3$	

So, z1 is the self-attention vector for the first word of the input sequence "Action gets results". We can get the vectors for the rest of the words in the input sequence in the same fashion:

Word	q vector	k vector	v vector	score	score / 8	Softmax	Softmax * v	Sum
Action		k_1	v_1	$q_2 . k_1$	$q_2 . k_1 / 8$	x_{21}	$x_{21} * v_1$	
gets	q_2	k_2	v_2	$q_2 . k_2$	$q_2 . k_2 / 8$	x_{22}	$x_{22} * v_2$	z_2
results		k_3	v_3	$q_2 . k_3$	$q_2 . k_3 / 8$	x_{23}	$x_{23} * v_3$	

Word	q vector	k vector	v vector	score	score / 8	Softmax	Softmax * v	Sum
Action		k_1	v_1	$q_3 . k_1$	$q_3 . k_1 / 8$	x_{31}	$x_{31} * v_1$	
gets		k_2	v_2	$q_3 . k_2$	$q_3 . k_2 / 8$	x_{32}	$x_{32} * v_2$	
results	q_3	k_3	v_3	$q_3 . k_3$	$q_3 . k_3 / 8$	x_{33}	$x_{33} * v_3$	z_3

In the Transformer's architecture, self-attention is computed multiple times, in parallel and independently. This process is referred to as Multi-head Attention. The outputs concatenate and linearly transform, as depicted in the figure below:



Multi-Head Attention

Limitations of the Transformer

Transformer architecture NLP is undoubtedly a huge improvement over the RNN based seq2seq models. But it comes with its own share of limitations:

- Attention can only deal with fixed-length text strings. The text has to be split into a certain number of segments or chunks before being fed into the system as input
- This chunking of text causes **context fragmentation**. For example, if a sentence is split from the middle, then a significant amount of context is lost. In other words, the text is split without respecting the sentence or any other semantic boundary