

UNIT – V

Deep Learning for Natural Language Processing

Computational representation of language, one-hot representation of words, word vectors – the skip-gram word2vec model, The CBOW word2vec model, word vector arithmetic, RNN, LSTM.

What is word embedding?

Word embedding or word vector is an approach with which we represent documents and words. It is defined as a numeric vector input that allows words with similar meanings to have the same representation. It can approximate meaning and represent a word in a lower dimensional space. These can be trained much faster than the hand-built models that use graph embeddings like WordNet.

The problem

Given a supervised learning task to predict which tweets are about real disasters and which ones are not (classification). Here the independent variable would be the tweets (text) and the target variable would be the binary values (1: Real Disaster, 0: Not real Disaster).

Now, Machine Learning and Deep Learning algorithms only take numeric input. So, how do we convert tweets to their numeric values? We will dive deep into the techniques to solve such problems, but first let's look at the solution provided by word embedding.

The solution

Word Embeddings in NLP is a technique where individual words are represented as real-valued vectors in a lower-dimensional space and captures inter-word semantics. Each word is represented by a real-valued vector with tens or hundreds of dimensions.

Different Types of Word Embeddings

The different types of word embeddings can be broadly classified into two categories -

1. Frequency-based Embedding
2. Prediction-based Embedding

Let us try to understand each of these methods in detail.

Frequency-based Embedding

There are generally three types of vectors that we encounter under this category.

1. Count Vector
2. TF-IDF Vector
3. Co-Occurrence Vector

Prediction-based Embedding

Pre-requisite: This section assumes that you have a working knowledge of how a neural network works and the mechanisms by which weights in an NN are updated. If you are new to Neural Network.

So far, we have seen deterministic methods to determine word vectors. But these methods proved to be limited in their word representations so word2vec is introduced to the NLP community. These methods were prediction based in the sense that they provided probabilities to the words and proved to be state of the art for tasks like word analogies and word similarities. They were also able to achieve tasks like King - man + woman = Queen, which was considered a result almost magical. So let us look at the word2vec model used as of today to generate word vectors.

Word2vec is not a single algorithm but a combination of two techniques – **CBOW(Continuous bag of words) and Skip-gram model**. Both of these are shallow neural networks which map word(s) to the target variable which is also a word(s). Both of these techniques learn weights which act as word vector representations. Let us discuss both these methods separately and gain intuition into their working.

CBOW (Continuous Bag of words)

The way CBOW work is that it tends to predict the probability of a word given a context. A context may be a single word or a group of words. But for simplicity, I will take a single context word and try to predict a single target word.

Suppose, we have a corpus $C = \text{"Hey, this is sample corpus using only one context word."}$ and we have defined a context window of 1. This corpus may be converted into a training set for a CBOW model as follow. The input is shown below. The matrix on the right in the below image contains the one-hot encoded from of the input on the left.

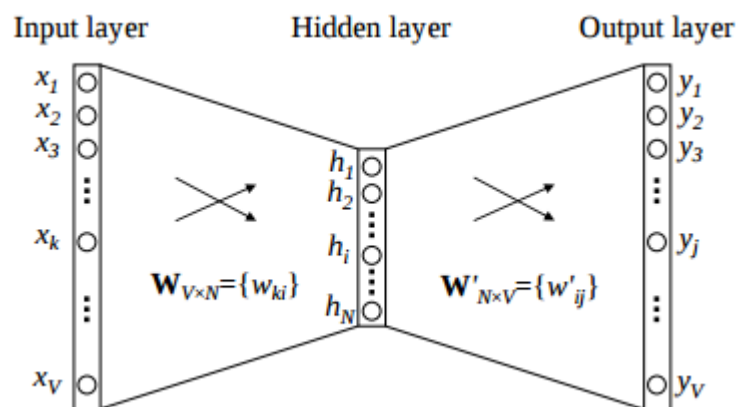
Input	Output		Hey	This	is	sample	corpus	using	only	one	context	word
Hey	this	Datapoint 1	1	0	0	0	0	0	0	0	0	0
this	hey	Datapoint 2	0	1	0	0	0	0	0	0	0	0
is	this	Datapoint 3	0	0	1	0	0	0	0	0	0	0
is	sample	Datapoint 4	0	0	1	0	0	0	0	0	0	0
sample	is	Datapoint 5	0	0	0	1	0	0	0	0	0	0
sample	corpus	Datapoint 6	0	0	0	1	0	0	0	0	0	0
corpus	sample	Datapoint 7	0	0	0	0	1	0	0	0	0	0
corpus	using	Datapoint 8	0	0	0	0	1	0	0	0	0	0
using	corpus	Datapoint 9	0	0	0	0	0	1	0	0	0	0
using	only	Datapoint 10	0	0	0	0	0	1	0	0	0	0
only	using	Datapoint 11	0	0	0	0	0	0	1	0	0	0
only	one	Datapoint 12	0	0	0	0	0	0	1	0	0	0
one	only	Datapoint 13	0	0	0	0	0	0	0	1	0	0
one	context	Datapoint 14	0	0	0	0	0	0	0	1	0	0
context	one	Datapoint 15	0	0	0	0	0	0	0	0	1	0
context	word	Datapoint 16	0	0	0	0	0	0	0	0	1	0
word	context	Datapoint 17	0	0	0	0	0	0	0	0	0	1

The target for a single datapoint say Datapoint 4 is shown as below

Hey	this	is	sample	corpus	using	only	one	context	word
0	0	0	1	0	0	0	0	0	0

This matrix shown in the above image is sent into a shallow neural network with three layers: an input layer, a hidden layer and an output layer. The output layer is a softmax layer which is used to sum the probabilities obtained in the output layer to 1. Now let us see how the forward propagation will work to calculate the hidden layer activation.

Let us first see a diagrammatic representation of the CBOW model.



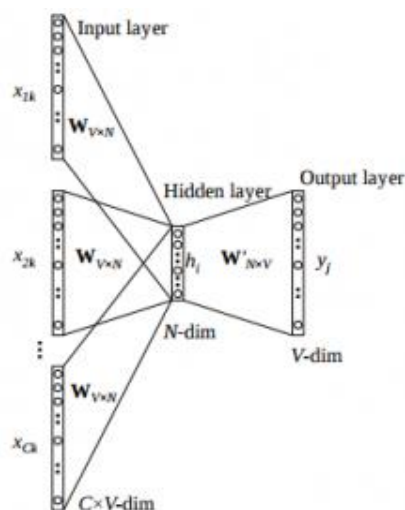
The matrix representation of the above image for a single data point is below.

Context										Input-Hidden Weight				Hidden Activation			
										1	2	3	4				
										5	6	7	8				
										9	10	11	12				
										13	14	15	16				
										17	18	19	20				
										21	22	23	24				
										25	26	27	28				
										29	30	31	32				
										33	34	35	36				
										37	38	39	40				
C1	this	0	1	0	0	0	0	0	0					5	6	7	8

The flow is as follows:

1. The input layer and the target, both are one-hot encoded of size $[1 \times V]$. Here $V=10$ in the above example.
2. There are two sets of weights. one is between the input and the hidden layer and second between hidden and output layer. Input-Hidden layer matrix size $= [V \times N]$, hidden-Output layer matrix size $= [N \times V]$: Where N is the number of dimensions we choose to represent our word in. It is arbitrary and a hyper-parameter for a Neural Network. Also, N is the number of neurons in the hidden layer. Here, $N=4$.
3. There is a no activation function between any layers.(More specifically, I am referring to linear activation)
4. The input is multiplied by the input-hidden weights and called hidden activation. It is simply the corresponding row in the input-hidden matrix copied.
5. The hidden input gets multiplied by hidden- output weights and output is calculated.
6. Error between output and target is calculated and propagated back to re-adjust the weights.
7. The weight between the hidden layer and the output layer is taken as the word vector representation of the word.

We saw the above steps for a single context word. Now, what about if we have multiple context words? The image below describes the architecture for multiple context words.



Below is a matrix representation of the above architecture for an easy understanding.

																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																					</
--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	----

2. Training a CBOW from scratch can take forever if not properly optimized.

Skip – Gram model

Skip – gram follows the same topology as of CBOW. It just flips CBOW’s architecture on its head. The aim of skip-gram is to predict the context given a word. Let us take the same corpus that we built our CBOW model on. C=“Hey, this is sample corpus using only one context word.” Let us construct the training data.

Input	Output(Context1)	Output(Context2)
Hey	this	<padding>
this	Hey	is
is	this	sample
sample	is	corpus
corpus	sample	corpus
using	corpus	only
only	using	one
one	only	context
context	one	word
word	context	<padding>

The input vector for skip-gram is going to be similar to a 1-context CBOW model. Also, the calculations up to hidden layer activations are going to be the same. The difference will be in the target variable. Since we have defined a context window of 1 on both the sides, there will be **“two” one hot encoded target variables** and **“two” corresponding outputs** as can be seen by the blue section in the image.

Two separate errors are calculated with respect to the two target variables and the two error vectors obtained are added element-wise to obtain a final error vector which is propagated back to update the weights.

The weights between the input and the hidden layer are taken as the word vector representation after training. The loss function or the objective is of the same type as of the CBOW model.

The skip-gram architecture is shown below.

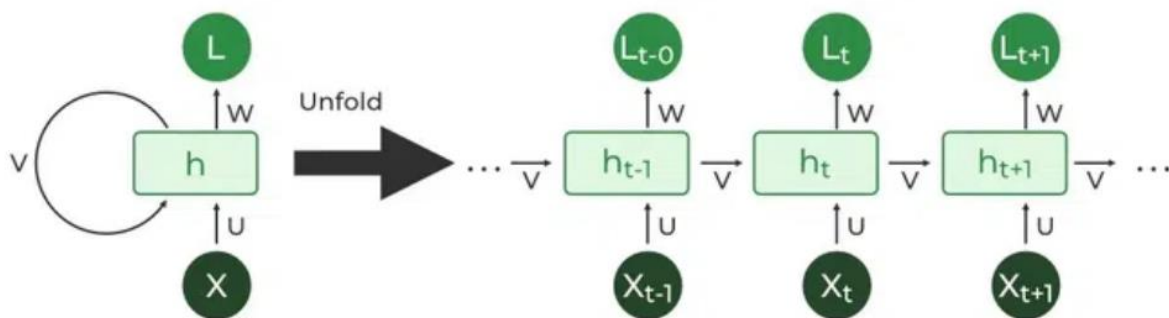
4. Each row of the blue matrix is converted into its softmax probabilities individually as shown in the green box.
5. The grey matrix contains the one hot encoded vectors of the two context words(target).
6. Error is calculated by subtracting the first row of the grey matrix(target) from the first row of the green matrix(output) element-wise. This is repeated for the next row. Therefore, for **n** target context words, we will have **n** error vectors.
7. Element-wise sum is taken over all the error vectors to obtain a final error vector.
8. This error vector is propagated back to update the weights.

Advantages of Skip-Gram Model:

1. Skip-gram model can capture two semantics for a single word. i.e it will have two vector representations of Apple. One for the company and other for the fruit.
2. Skip-gram with negative sub-sampling outperforms every other method generally.

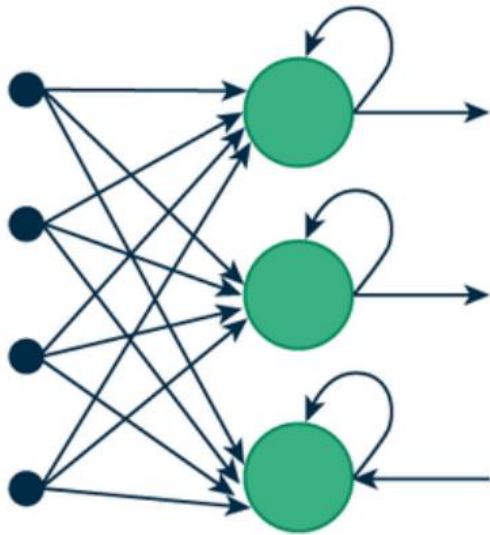
Recurrent Neural Network:

Recurrent Neural Network (RNN) is a type of Neural Network 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. Still, 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.

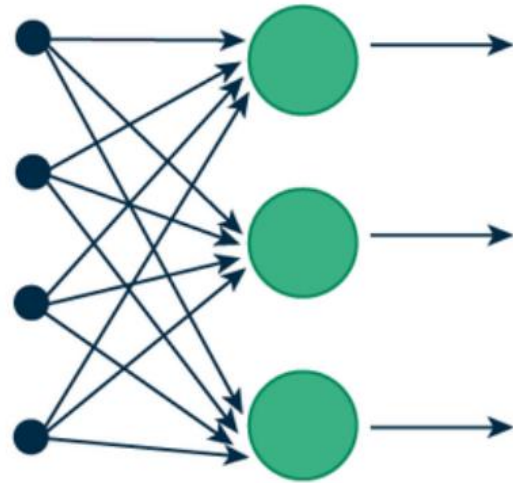


How RNN differs from Feedforward Neural Network?

Artificial neural networks that do not have looping nodes are called feed forward neural networks. Because all information is only passed forward, this kind of neural network is also referred to as a multi-layer neural network. Information moves from the input layer to the output layer – if any hidden layers are present – unidirectionally in a feedforward neural network. These networks are appropriate for image classification tasks, for example, where input and output are independent. Nevertheless, their inability to retain previous inputs automatically renders them less useful for sequential data analysis.

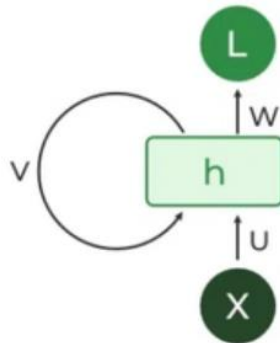


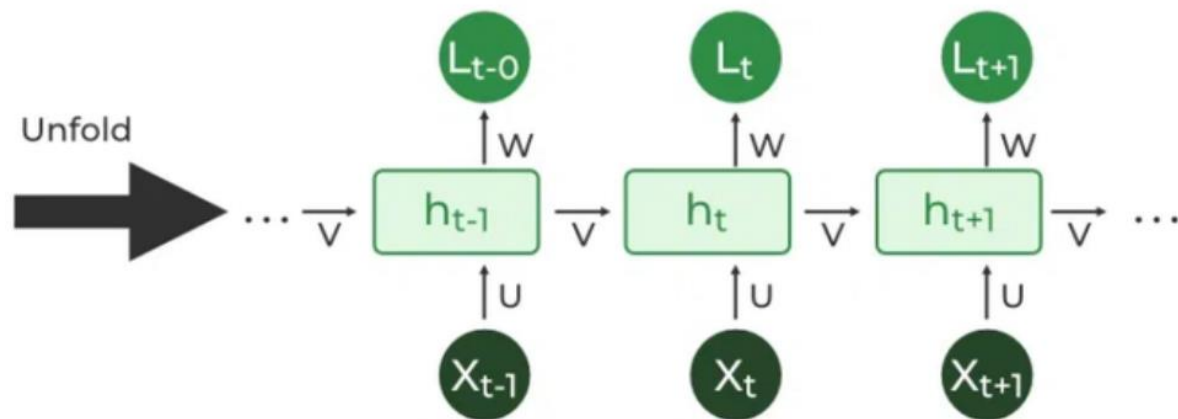
(a) Recurrent Neural Network



(b) Feed-Forward Neural Network

The fundamental processing unit in a Recurrent Neural Network (RNN) is a Recurrent Unit, which is not explicitly called a “Recurrent Neuron.” This unit has the unique ability to maintain a hidden state, allowing the network to capture sequential dependencies by remembering previous inputs while processing. Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) versions improve the RNN’s ability to handle long-term dependencies.

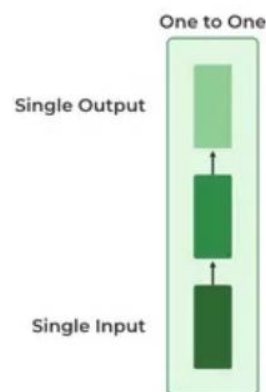




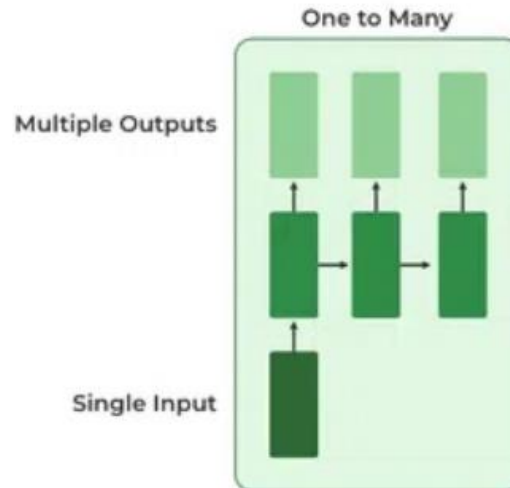
Types of RNN:

There are four types of RNNs based on the number of inputs and outputs in the network.

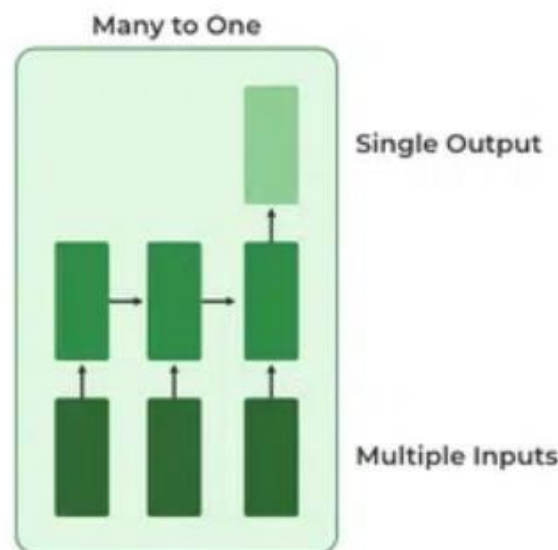
1. One to One: This type of RNN behaves the same as any simple Neural network it is also known as Vanilla Neural Network. In this Neural network, there is only one input and one output.



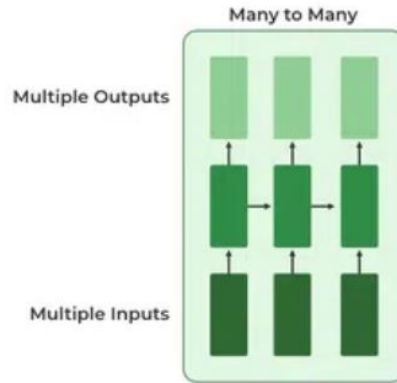
2. One To Many: In this type of RNN, there is one input and many outputs associated with it. One of the most used examples of this network is Image captioning where given an image we predict a sentence having Multiple words.



3. Many to One: In this type of network, Many inputs are fed to the network at several states of the network generating only one output. This type of network is used in the problems like sentimental analysis. Where we give multiple words as input and predict only the sentiment of the sentence as output.



4. Many to Many: In this type of neural network, there are multiple inputs and multiple outputs corresponding to a problem. One Example of this Problem will be language translation. In language translation, we provide multiple words from one language as input and predict multiple words from the second language as output.



Recurrent Neural Network Architecture:

RNNs have the same input and output architecture as any other deep neural architecture. However, differences arise in the way information flows from input to output. Unlike Deep neural networks where we have different weight matrices for each Dense network in RNN, the weight across the network remains the same. It calculates state hidden state H_i for every input X_i . By using the following formulas:

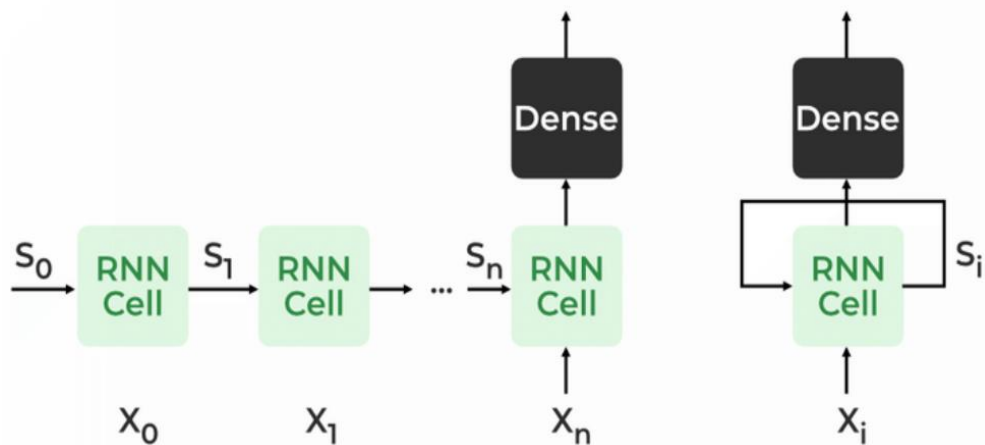
$$h = \sigma(UX + Wh_{-1} + B)$$

$$Y = O(Vh + C)$$

Hence

$$Y = f(X, h, W, U, V, B, C)$$

Here S is the State matrix which has element s_i as the state of the network at timestep i . The parameters in the network are W, U, V, c, b which are shared across timestep.



The Recurrent Neural Network consists of multiple fixed activation function units, one for each time step. Each unit has an internal state which is called the hidden state of the unit.

This hidden state signifies the past knowledge that the network currently holds at a given time step. This hidden state is updated at every time step to signify the change in the knowledge of the network about the past. The hidden state is updated using the following recurrence relation:-

The formula for calculating the current state:

$$h_t = f(h_{t-1}, x_t)$$

where,

- h_t -> current state
- h_{t-1} -> previous state
- x_t -> input state

Formula for applying Activation function(tanh)

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$$

where,

- w_{hh} -> weight at recurrent neuron
- w_{xh} -> weight at input neuron

The formula for calculating output:

$$y_t = W_{hy}h_t$$

- Y_t -> output
- W_{hy} -> weight at output layer

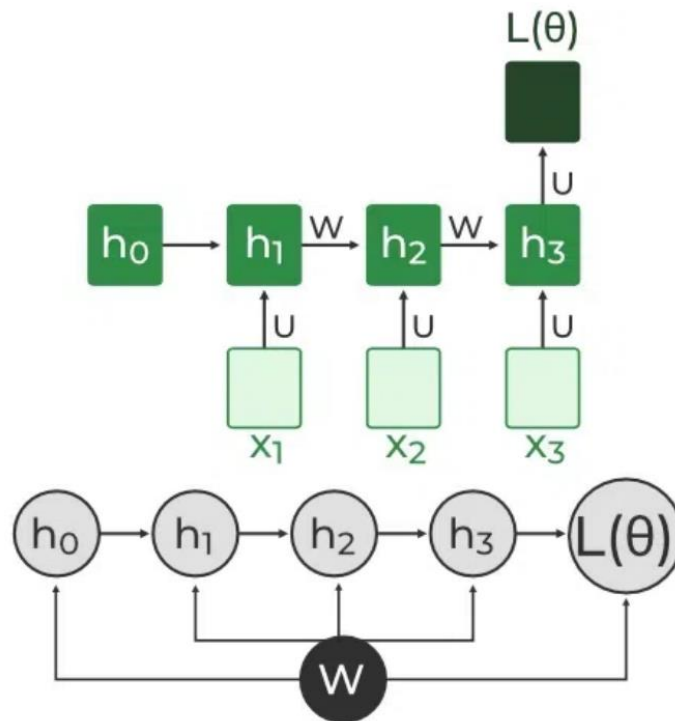
These parameters are updated using Backpropagation. However, since RNN works on sequential data here we use an updated backpropagation which is known as Backpropagation through time.

Backpropagation Through Time (BPTT):

In RNN the neural network is in an ordered fashion and since in the ordered network each variable is computed one at a time in a specified order like first h_1 then h_2 then h_3 so on. Hence we will apply backpropagation throughout all these hidden time states sequentially.

- $L(\theta)$ (loss function) depends on h_3
- h_3 in turn depends on h_2 and W
- h_2 in turn depends on h_1 and W
- h_1 in turn depends on h_0 and W

- where h_0 is a constant starting state.



$$\frac{\partial \mathbf{L}(\theta)}{\partial W} = \sum_{t=1}^T \frac{\partial \mathbf{L}(\theta)}{\partial W}$$

For simplicity of this equation, we will apply backpropagation on only one row

$$\frac{\partial L(\theta)}{\partial W} = \frac{\partial L(\theta)}{\partial h_3} \frac{\partial h_3}{\partial W}$$

we know $h_3 = \sigma(W h_2 + b)$

And In such an ordered network, we can't compute $\frac{\partial h_3}{\partial W}$ by simply treating h_3 as a constant

because as it also depends on W . the total derivative $\frac{\partial h_3}{\partial W}$ has two parts:

- 1. Explicit:** $\frac{\partial h_3}{\partial W}$ treating all other inputs as constant.
- 2. Implicit:** Summing over all indirect paths from h_3 to W .

Let us see how to do this

$$\begin{aligned}
\frac{\partial h_3}{\partial W} &= \frac{\partial h_3^+}{\partial W} + \frac{\partial h_3}{\partial h_2} \frac{\partial h_2}{\partial W} \\
&= \frac{\partial h_3^+}{\partial W} + \frac{\partial h_3}{\partial h_2} \left[\frac{\partial h_2^+}{\partial W} + \frac{\partial h_2}{\partial h_1} \frac{\partial h_1}{\partial W} \right] \\
&= \frac{\partial h_3^+}{\partial W} + \frac{\partial h_3}{\partial h_2} \frac{\partial h_2^+}{\partial W} + \frac{\partial h_3}{\partial h_2} \frac{\partial h_2}{\partial h_1} \left[\frac{\partial h_1^+}{\partial W} \right]
\end{aligned}$$

For simplicity, we will short-circuit some of the paths

$$\frac{\partial h_3}{\partial W} = \frac{\partial h_3^+}{\partial W} + \frac{\partial h_3}{\partial h_2} \frac{\partial h_2^+}{\partial W} + \frac{\partial h_3}{\partial h_1} \frac{\partial h_1^+}{\partial W}$$

Finally, we have

$$\frac{\partial L(\theta)}{\partial W} = \frac{\partial L(\theta)}{\partial h_3} \cdot \frac{\partial h_3}{\partial W}$$

Where

$$\frac{\partial h_3}{\partial W} = \sum_{k=1}^3 \frac{\partial h_3}{\partial h_k} \cdot \frac{\partial h_k}{\partial W}$$

Hence,

$$\frac{\partial L(\theta)}{\partial W} = \frac{\partial L(\theta)}{\partial h_3} \sum_{k=1}^3 \frac{\partial h_3}{\partial h_k} \cdot \frac{\partial h_k}{\partial W}$$

This algorithm is called backpropagation through time (BPTT) as we backpropagate over all previous time steps.

Issues of Standard RNNs :

1. **Vanishing Gradient:** Text generation, machine translation, and stock market prediction are just a few examples of the time-dependent and sequential data problems that can be modelled with recurrent neural networks. You will discover, though, that the gradient problem makes training RNN difficult.

2. **Exploding Gradient:** An Exploding Gradient occurs when a neural network is being trained and the slope tends to grow exponentially rather than decay. Large error gradients that build up during training lead to very large updates to the neural network model weights, which is the source of this issue.

Training through RNN:

1. A single-time step of the input is provided to the network.
2. Then calculate its current state using a set of current input and the previous state.
3. The current h_t becomes h_{t-1} for the next time step.

4. One can go as many time steps according to the problem and join the information from all the previous states.
5. Once all the time steps are completed the final current state is used to calculate the output.
6. The output is then compared to the actual output i.e the target output and the error is generated.
7. The error is then back-propagated to the network to update the weights and hence the network (RNN) is trained using Backpropagation through time.

Advantages and Disadvantages of Recurrent Neural Network:

Advantages:

1. An RNN remembers each and every piece of information through time. It is useful in time series prediction only because of the feature to remember previous inputs as well. This is called Long Short Term Memory.
2. Recurrent neural networks are even used with convolutional layers to extend the effective pixel neighborhood.

Disadvantages:

1. Gradient vanishing and exploding problems.
2. Training an RNN is a very difficult task.
3. It cannot process very long sequences if using tanh or relu as an activation function.

Applications of Recurrent Neural Network:

1. Language Modelling and Generating Text
2. Speech Recognition
3. Machine Translation
4. Image Recognition, Face detection
5. Time series Forecasting

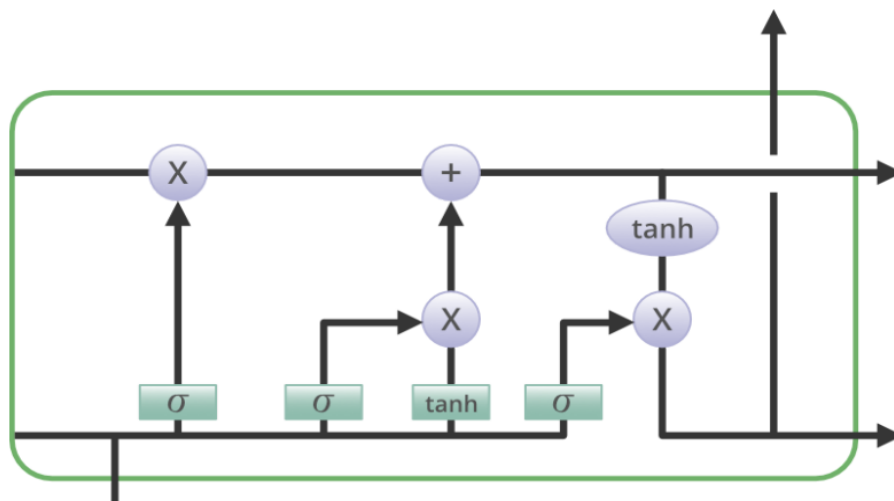
Long Short Term Memory:

Long Short-Term Memory is an improved version of recurrent neural network designed by Hochreiter & Schmidhuber. LSTM is well-suited for sequence prediction tasks and excels in capturing long-term dependencies. Its applications extend to tasks involving time series and sequences. LSTM's strength lies in its ability to grasp the order dependence crucial for solving intricate problems, such as machine translation and speech recognition.

A traditional RNN has a single hidden state that is passed through time, which can make it difficult for the network to learn long-term dependencies. LSTMs address this problem by introducing a memory cell, which is a container that can hold information for an extended period. LSTM networks are capable of learning long-term dependencies in sequential data, which makes them well-suited for tasks such as language translation, speech recognition, and time series forecasting. LSTMs can also be used in combination with other neural network architectures, such as Convolutional Neural Networks (CNNs) for image and video analysis. The memory cell is controlled by three gates: the input gate, the forget gate, and the output gate. These gates decide what information to add to, remove from, and output from the memory cell. The input gate controls what information is added to the memory cell. The forget gate controls what information is removed from the memory cell. And the output gate controls what information is output from the memory cell. This allows LSTM networks to selectively retain or discard information as it flows through the network, which allows them to learn long-term dependencies.

Architecture and Working of LSTM:

LSTM architecture has a chain structure that contains four neural networks and different memory blocks called cells.



Information is retained by the cells and the memory manipulations are done by the gates.

There are three gates –

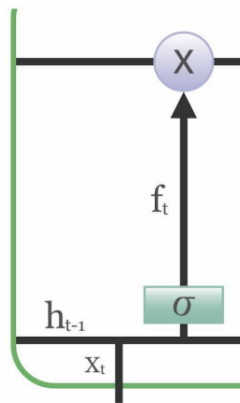
Forget Gate: The information that is no longer useful in the cell state is removed with the forget gate. Two inputs x (input at the particular time) and h (previous cell output) are fed

to the gate and multiplied with weight matrices followed by the addition of bias. The resultant is passed through an activation function which gives a binary output. If for a particular cell state the output is 0, the piece of information is forgotten and for output 1, the information is retained for future use. The equation for the forget gate is:

$$f_t = (W_f[h_{t-1}, x_t] + b_f)$$

where:

- W_f represents the weight matrix associated with the forget gate.
- $[h_{t-1}, x_t]$ denotes the concatenation of the current input and the previous hidden state.
- b_f is the bias with the forget gate.
- σ is the sigmoid activation function.



Input gate:

The addition of useful information to the cell state is done by the input gate. First, the information is regulated using the sigmoid function and filter the values to be remembered similar to the forget gate using inputs h_{t-1} and x_t . Then, a vector is created using tanh function that gives an output from -1 to +1, which contains all the possible values from h_{t-1} and x_t . At last, the values of the vector and the regulated values are multiplied to obtain the useful information. The equation for the input gate is:

$$i_t = (W_i[h_{t-1}, x_t] + b_i)$$

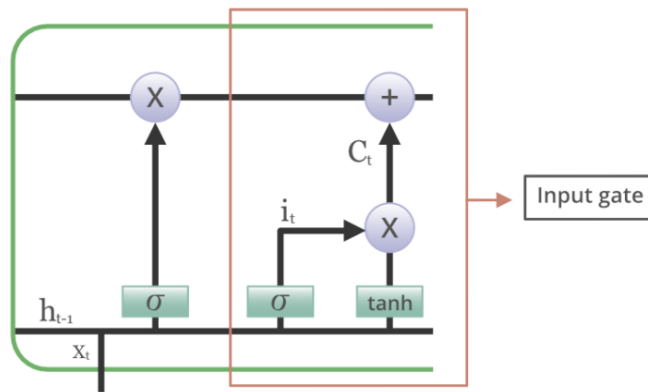
$$\hat{C}_t = \tanh(W_c[h_{t-1}, x_t] + b_c)$$

We multiply the previous state by f_t , disregarding the information we had previously chosen to ignore. Next, we include $i_t * C_t$. This represents the updated candidate values, adjusted for the amount that we chose to update each state value.

$$C_t = f_t C_{t-1} + i_t \tilde{C}_t$$

where

- \odot denotes element-wise multiplication
- \tanh is tanh activation function



Output gate: The task of extracting useful information from the current cell state to be presented as output is done by the output gate. First, a vector is generated by applying tanh function on the cell. Then, the information is regulated using the sigmoid function and filter by the values to be remembered using inputs h_{t-1} and x_t . At last, the values of the vector and the regulated values are multiplied to be sent as an output and input to the next cell. The equation for the output gate is:

$$o_t = (W_o[h_{t-1}, x_t] + b_o)$$

