

UNIT-IV

Recurrent Neural Network

CONTENTS

- ❖ Introduction to RNN
- ❖ Comparison of FFNN and RNN
- ❖ Architecture of RNN
- ❖ How RNN Works
- ❖ Activation Function of RNN
- ❖ Mathematical Representation of RNN
- ❖ Types of RNN
- ❖ Applications of RNN

INTRODUCTION

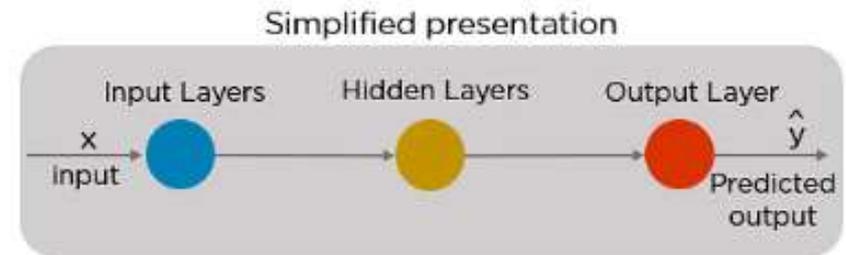
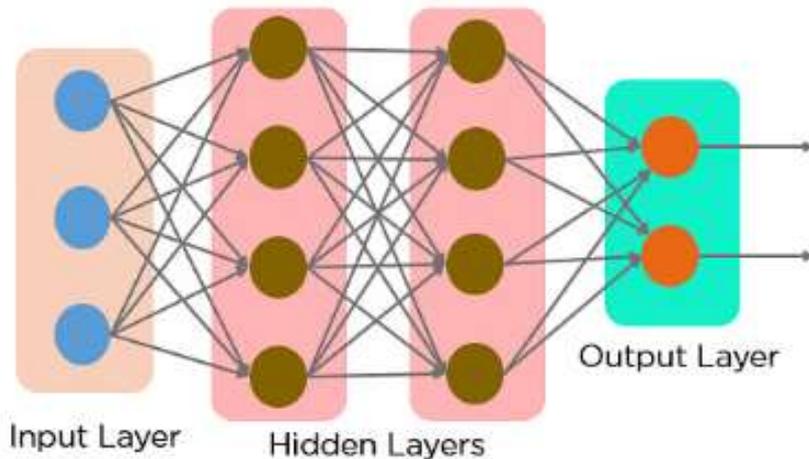
- **Q:** Ever wonder how chatbots understand your questions or how apps like Siri and voice search can decipher your spoken requests?
- **Ans:** The secret weapon behind these impressive feats is a type of artificial intelligence called **Recurrent Neural Networks (RNNs)**.
- **Recurrent Neural networks** imitate the function of the human brain in the fields of **Data science, Artificial intelligence, machine learning, and deep learning**, allowing **computer programs** to **recognize patterns and solve common issues**.
- **Recurrent neural network (RNN)** is a type of **artificial neural network** that is used to process sequential data.

- In **Recurrent Neural Network(RNN)** 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**.
- In this Network is commonly used in speech recognition and natural language processing. Recurrent neural networks **recognize data's sequential characteristics** and use **patterns** to predict the next likely scenario.
- RNN use cases tend to be **connected to language models** in which knowing the next letter in a word or **the next word in a sentence** is predicated on the data that comes before it.

COMPARING FFNN AND RNN

- **FFNN**: A feed-forward neural network has **only one route of information flow**: from **the input layer to the output layer**, passing through the hidden layers.
- The **data flows across the network in a straight route**, never going through the same node twice.
- **A feed-forward neural network** can perform simple classification, regression, or recognition tasks, but it can't remember the previous input that it has processed.
- that's why FNNs are poor predictions of what will happen next because they have **no memory of the information they receive**.



- **RNN:** The information is in an RNN cycle via a loop. Before making a judgment, it evaluates the current input as well as what it has learned from past inputs.
- A recurrent neural network, on the other hand, may recall due to internal memory. It produces output, copies it, and then returns it to the network.

Why Recurrent Neural Networks?

Issues with Feedforward Neural Network

01
Unable to handle sequential data

02
Only current input is considered

03
Unable to memorize previous outputs

03
Is able to memorize previous outputs because of internal memory

02
Considers the current input along with previous inputs

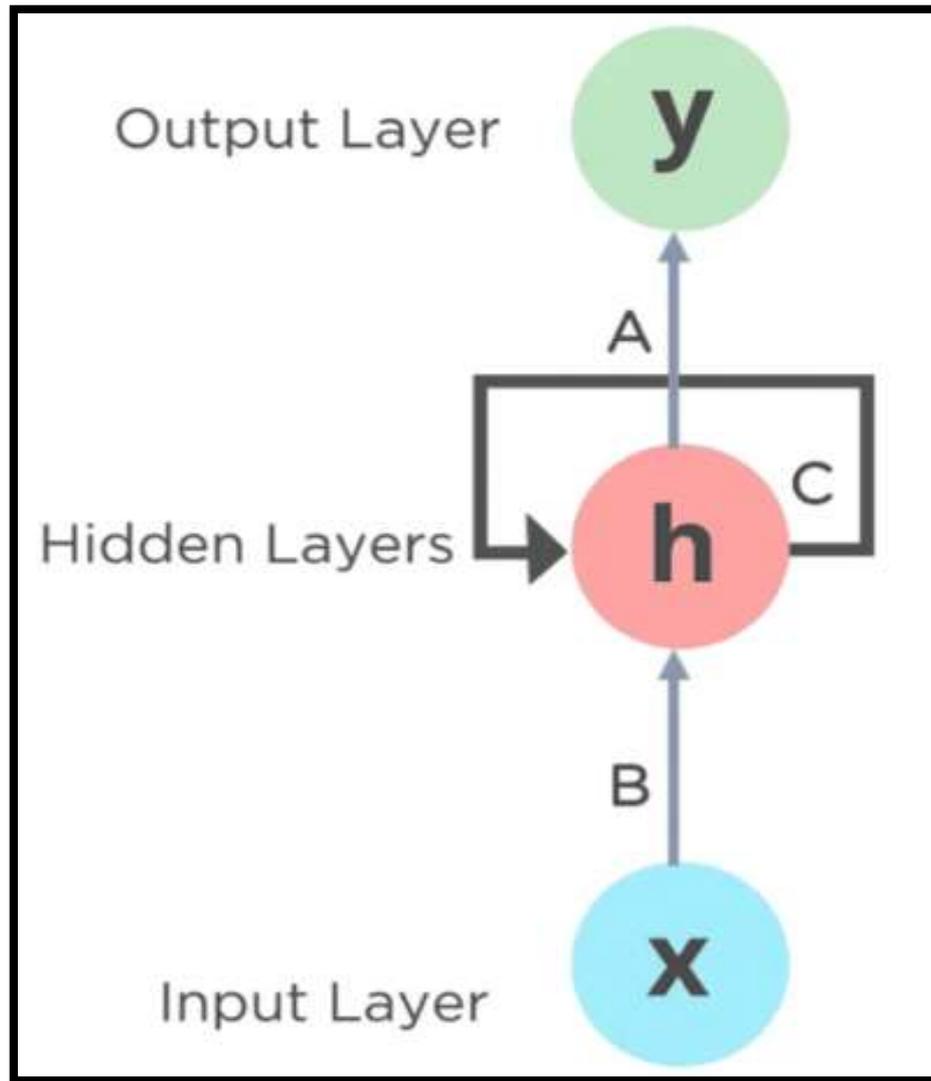
01
Is able to handle sequential data

Solution by Recurrent Neural Networks

ARCHITECTURE OF RNN

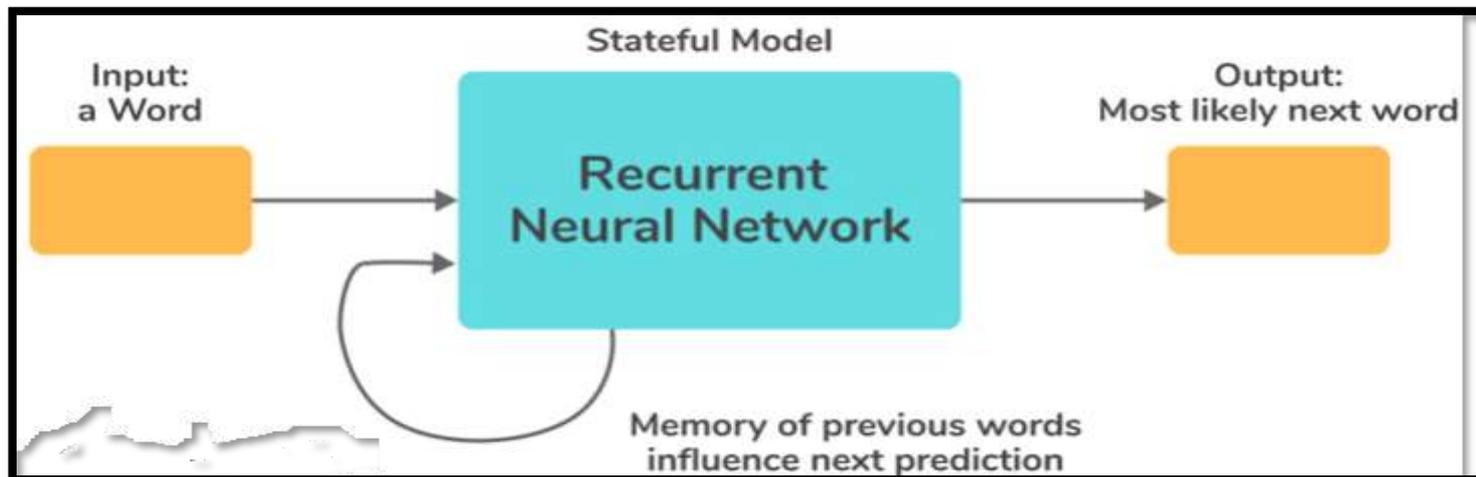
- A **recurrent neural network appears** very just like feed forward neural networks, **except it also has connections pointing backwards.**
- **It has 3 Layers.** Those are
 - **Input Layer**
 - **Hidden Layer and**
 - **Output Layer**
- RNNs are a type of neural network that has **hidden states and allows past outputs to be used as inputs.** They usually go like this:

ARCHITECTURE OF RNN

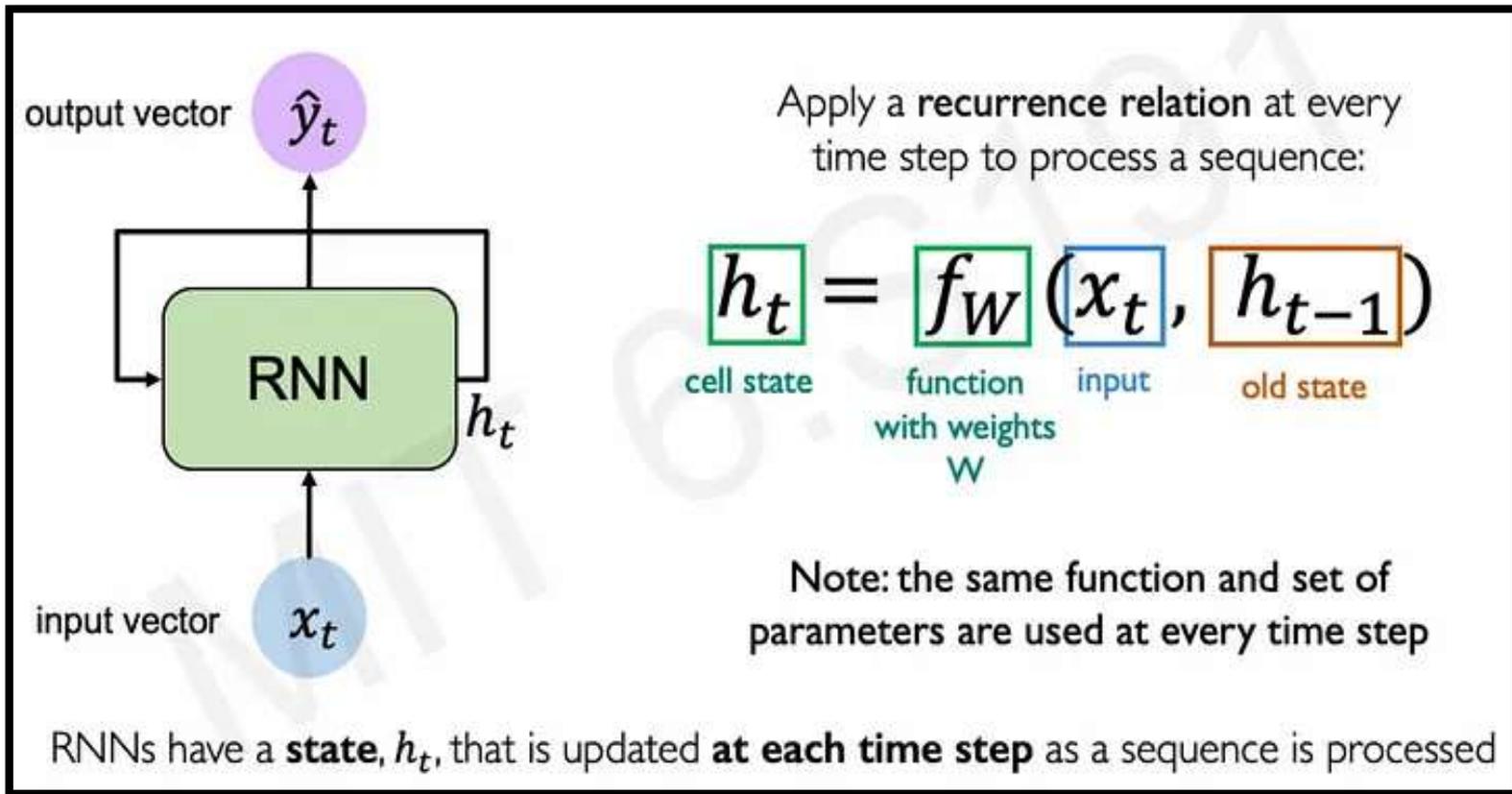


- The fundamental processing unit in a Recurrent Neural Network (RNN) is a Recurrent Unit, which is not explicitly called a “Recurrent Neuron.”
- This unit has the unique ability to maintain a hidden state, allowing the network to capture sequential dependencies by remembering previous inputs while processing.
- Input Layer: Here, “x” is the input layer, “h” is the hidden layer, and “y” is the output layer. A, B, and C are the network parameters used to improve the output of the model.

- Recurrent Layer: Normally a hidden layer or node has two parameters: bias and weight. But a recurrent node has three parameters: input, bias, and weight.
- At any given time t, the current input is a combination of input at $x(t)$ and $x(t-1)$. The output at any given time is fetched back to the network to improve on the output.

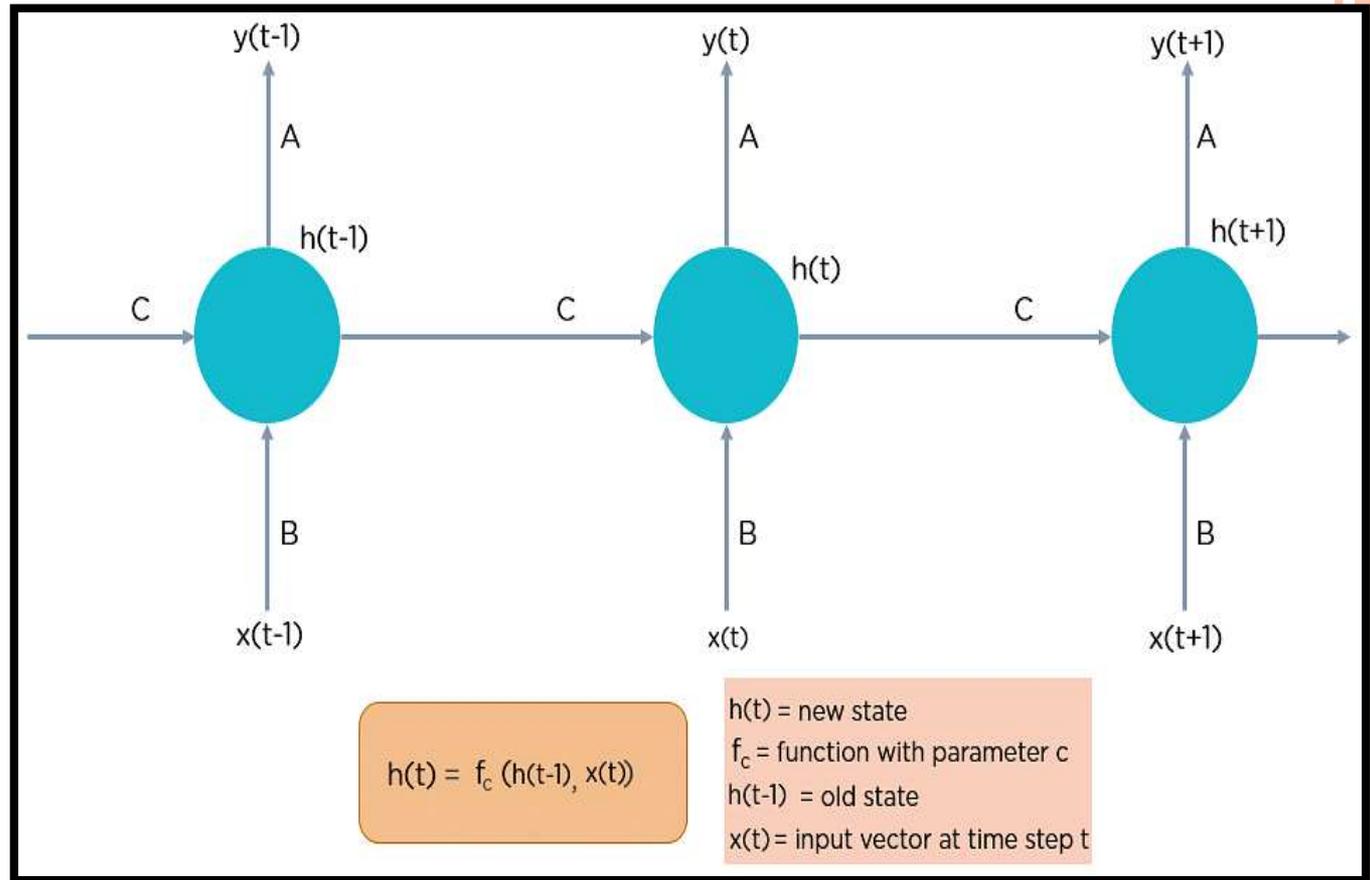
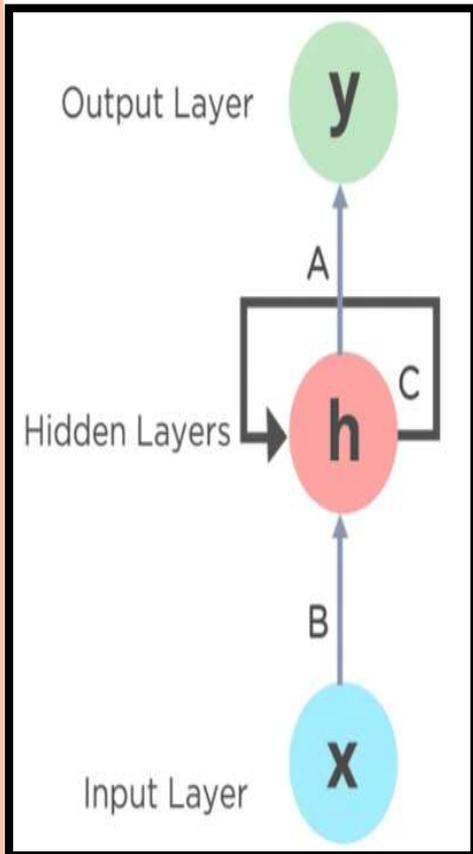


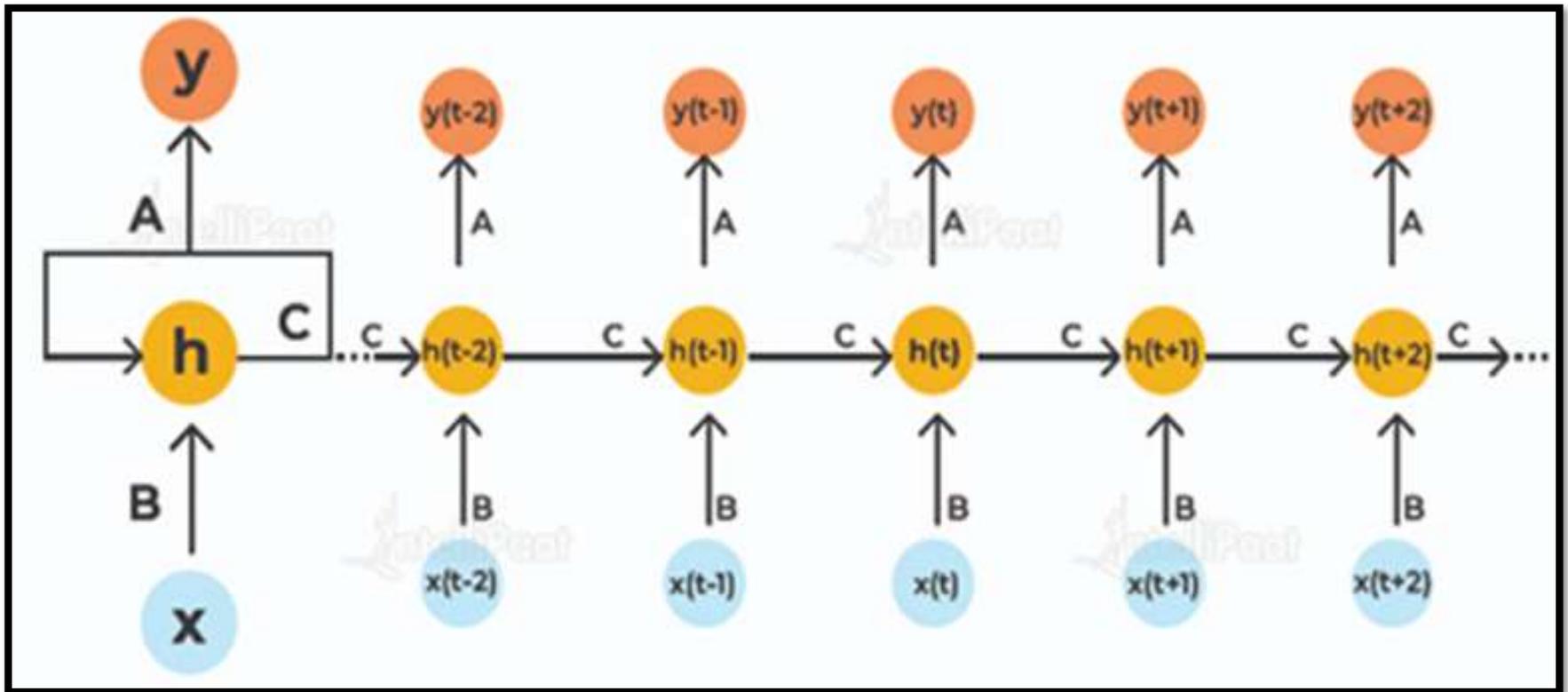
- In RNNs, the information cycles through the loop to the middle hidden layer.



HOW DOES RNN WORK

➤ Here the Unfolding RNN is :





- **Multiple hidden layers** can be found in the **middle layer h**, each with its **own activation functions, weights, and biases**.

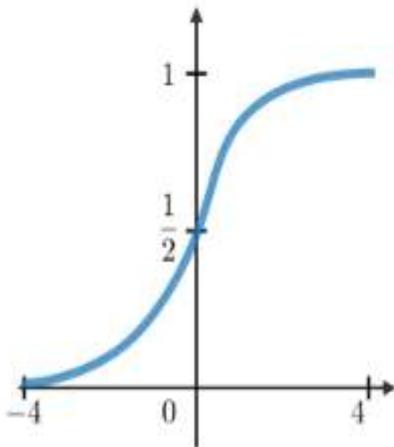
ACTIVATION FUNCTIONS USED IN RNN

- A **neuron's activation function** dictates **whether it should be turned on or off**. Nonlinear functions usually transform a neuron's output to a number between 0 and 1 or -1 and 1.

The following are some of the most commonly utilized functions:

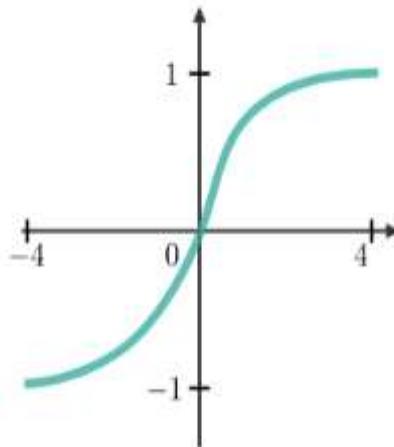
Sigmoid

$$g(z) = \frac{1}{1 + e^{-z}}$$



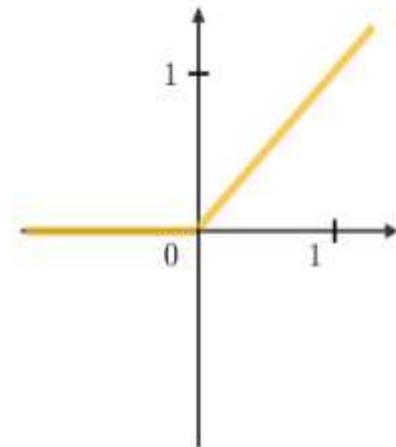
Tanh

$$g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$



RELU

$$g(z) = \max(0, z)$$



➤ **Sigmoid Function ($\sigma(x)$)**

Formula: $\sigma(x) = 1 / (1 + e^{(-x)})$

Behavior: Squishes any real number between 0 and 1.

➤ **Hyperbolic Tangent ($\tanh(x)$)**

Formula: $\tanh(x) = (e^x - e^{(-x)}) / (e^x + e^{(-x)})$

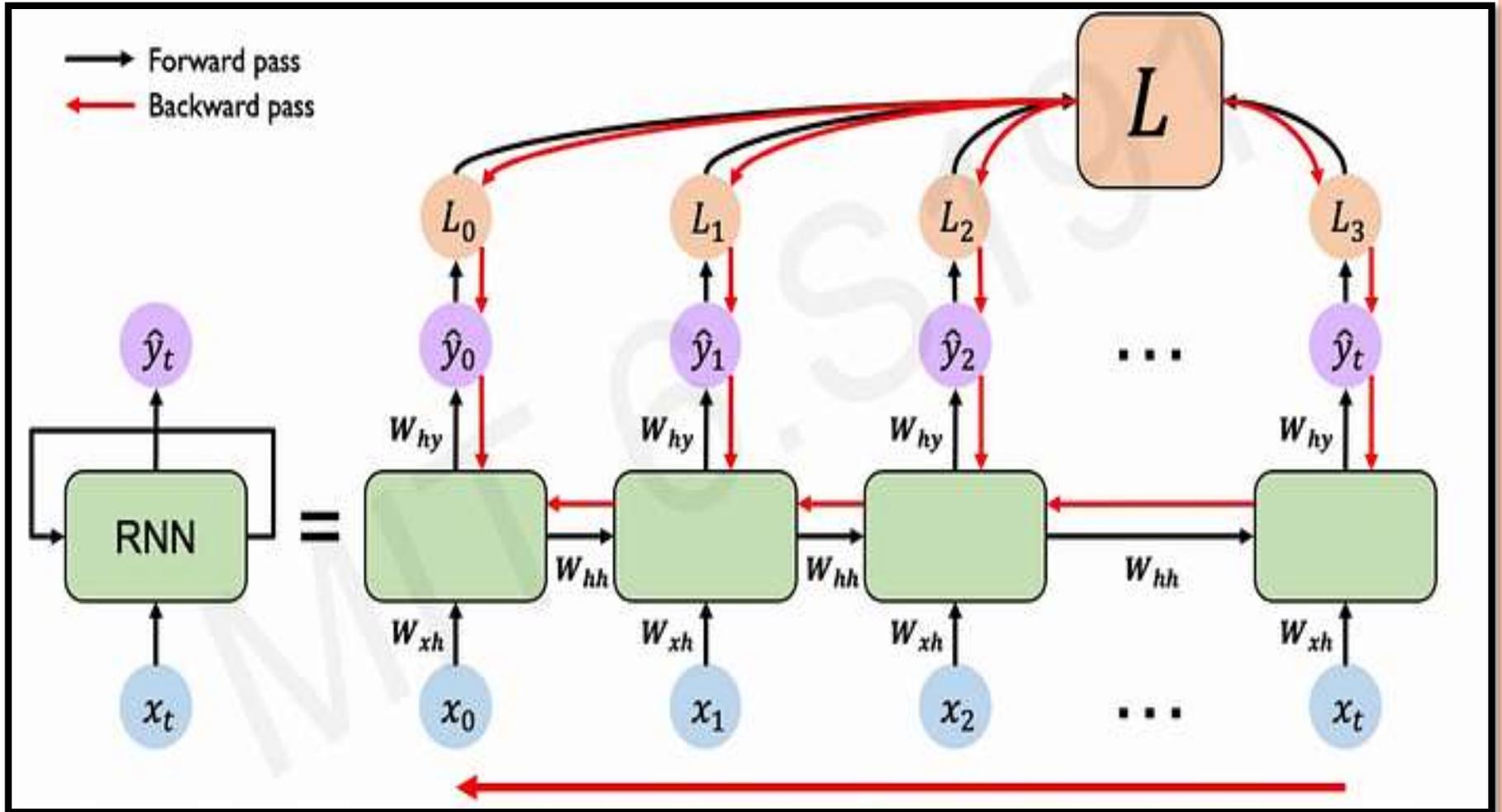
Behavior: Squeezes any real number between -1 and 1.

➤ **Rectified Linear Unit (ReLU)(x)**

Formula: $\text{ReLU}(x) = \max(0, x)$

Behavior: Outputs the input value if positive, otherwise outputs 0.

MATHEMATICAL REPRESENTATION



For calculating the current state –

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

h_t – current state

h_{t-1} – previous state

x_t – input state

In order to apply the activation function tanh, we have –

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

where:

w_{hh} -> weight of recurrent neuron and,

w_{xh} -> weight of the input neuron

The formula for calculating output:

$$y_t = W_{hy}h_t$$

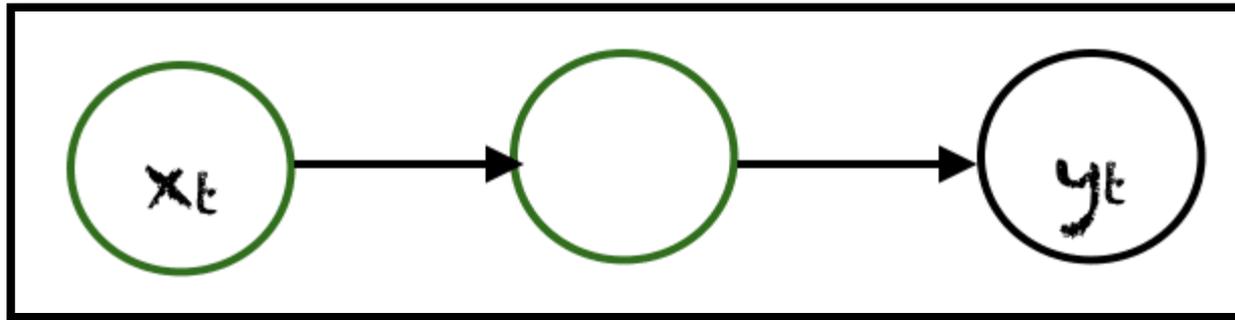
Y_t -> output

W_{hy} -> weight at output layer

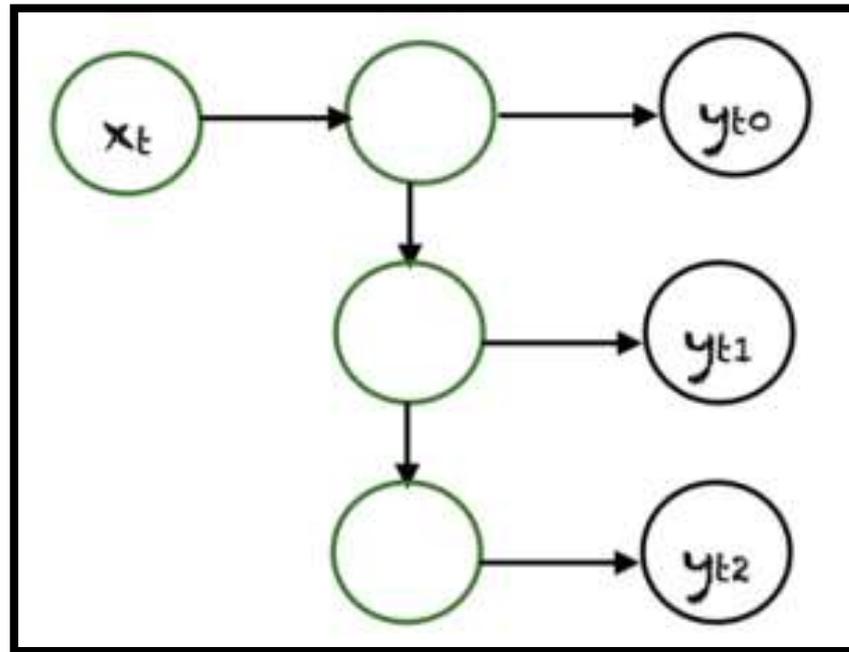
These parameters are updated using Back propagation. However, since **RNN works on sequential data** here we use an **updated back propagation** which is known as **Back propagation through time**.

TYPES OF RNN

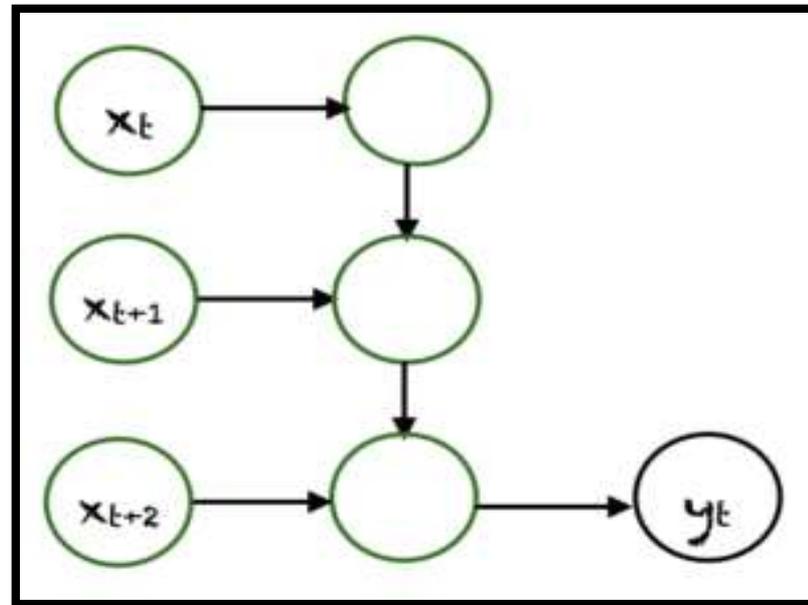
- There are **different types of recurrent neural networks** with varying architectures. Some examples are:
 - 1. **One to One**: Here, there is a **single (X_t, Y_t) pair**. Traditional neural networks **employ a one-to-one architecture**.



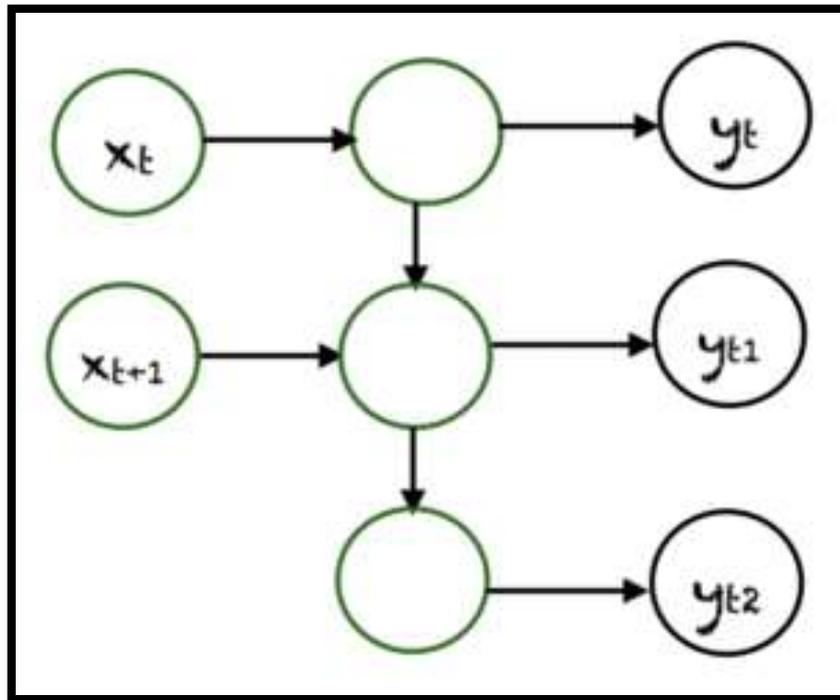
- 2. One to Many: In one-to-many networks, a single input at X_t can produce multiple outputs, e.g., (y_{t0}, y_{t1}, y_{t2}) .
- Music generation is an example area where one-to-many networks are employed.

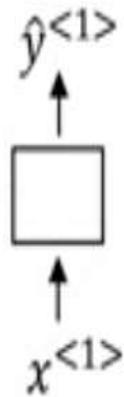


- 3. Many to One : In this case, many inputs from different time steps produce a single output.
- For example, (X_t, X_{t+1}, X_{t+2}) can produce a single output Y_t . Such networks are employed in **sentiment analysis or emotion detection**, where the **class label depends upon a sequence of words**

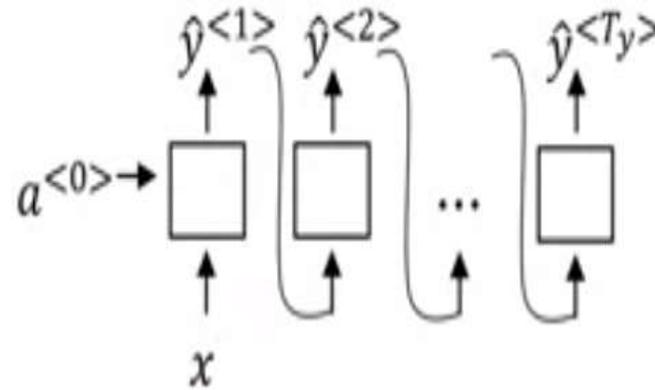


- 4. Many to Many : here are many possibilities for many-to-many. An example is shown above, where **two inputs produce three outputs**. Many-to-many networks are applied in **machine translation**, e.g., **English to French or vice versa translation systems**.

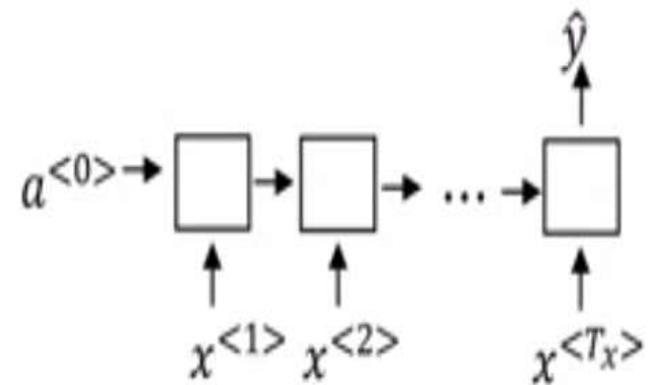




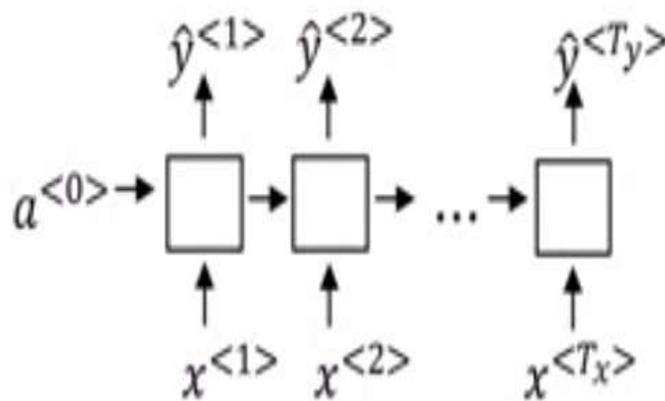
One to one



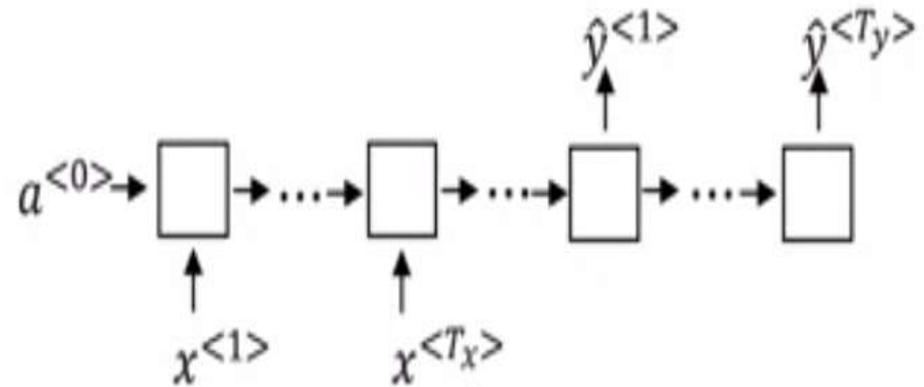
One to many



Many to one



Many to many



Many to many

APPLICATIONS OF RNN

- 1. Image Captioning: RNNs are used to caption an image by analyzing the activities present.



"A Dog catching a ball in mid air"

- 2. Time Series Prediction: Any time series problem, like predicting the prices of stocks in a particular month, can be solved using an RNN.
- 3. Natural Language Processing: Text mining and Sentiment analysis can be carried out using an RNN for Natural Language Processing (NLP).



Natural Language Processing

When it rains, look for rainbows.
When it's dark, look for stars.

Positive Sentiment

- **4. Machine Translation:** Given an **input in one language**, RNNs can be used to **translate the input into different languages as output**.
- Eg: **Google Translator**



Here the person is speaking in English and it is getting translated into Chinese, Italian, French, German and Spanish languages

Machine Translation



THANK YOU!

UNIT-IV

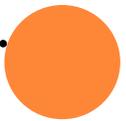
Encoder-Decoder Sequence-to-Sequence Architectures

CONTENTS

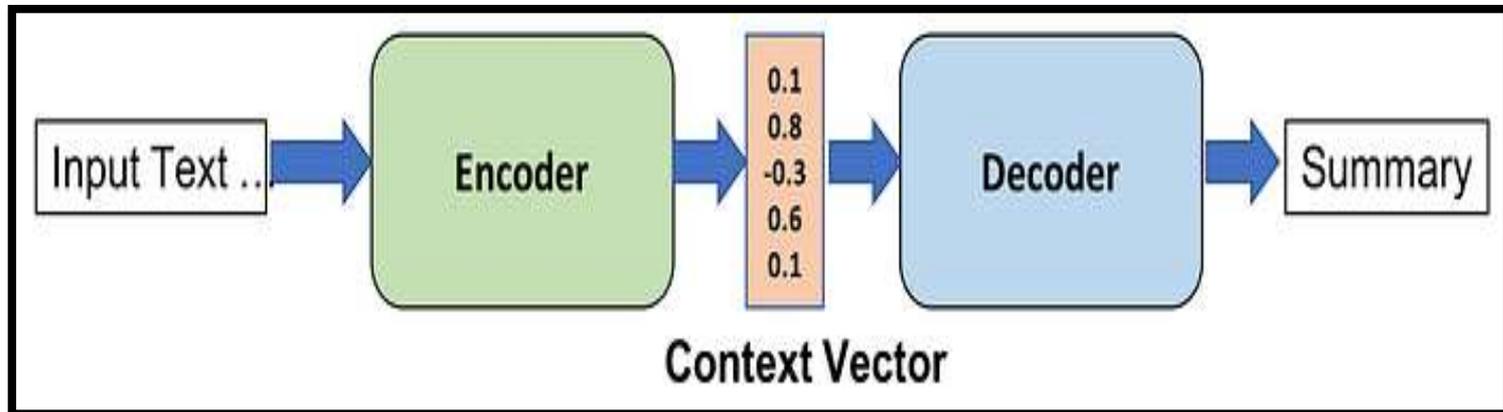
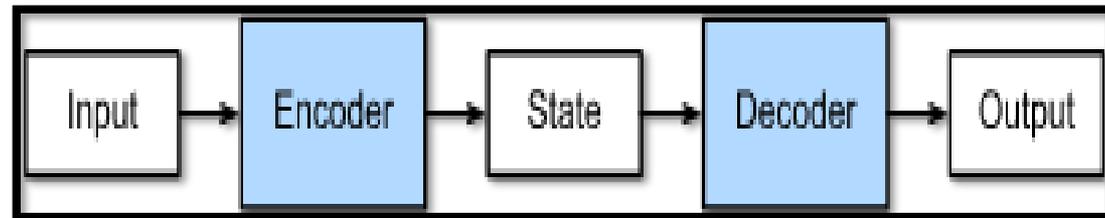
- ❖ Introduction
- ❖ Seq to Seq Model
- ❖ How Sequence Model Works
- ❖ Advantages
- ❖ Applications
- ❖ Applications of RNN

ENCODER-DECODER SEQUENCE TO SEQUENCE ARCHITECTURE

- In the field of Deep Learning, the **encoder-decoder architecture** is a widely-used framework for developing neural networks that can perform natural language processing (NLP) tasks such as **language translation, text summarization, and question-answering systems**, etc which require **sequence-to-sequence modeling**.
- This architecture involves a **two-stage process** where the **input data is first encoded** (using what is called an **encoder**) into a **Fixed-length numerical representation**, which is then **decoded** (using a **decoder**) to **produce an output** that matches the **desired format**.



- An **encoder-decoder** is a neural network architecture commonly used in **sequence-to-sequence (Seq2Seq)** models, particularly in tasks involving **natural language processing (NLP)** and **machine translation**. It consists of two main components: an **encoder** and a **decoder**.



- There are **3 main Blocks in the Encoder-Decoder model**
 - **Encoder**
 - **Hidden Vector**
 - **Decoder**
- The **Encoder will convert the input sequence into a single-dimensional vector** (hidden vector). The **decoder will convert the hidden vector into the output sequence.**



- 1. Encoder: The encoder takes an input sequence and processes it into a **Fixed-size representation** called the “**context vector**” or “**thought vector.**”
 - The **input sequence can be a sentence, paragraph, or any sequential data.** The encoder typically uses recurrent neural networks (RNNs) such as **LSTM (Long Short-Term Memory)** to **capture the sequential dependencies of the input.**
 - It processes the **input sequence step by step and summarizes the information in the context vector,** which aims to capture the essential information of the input.
- 

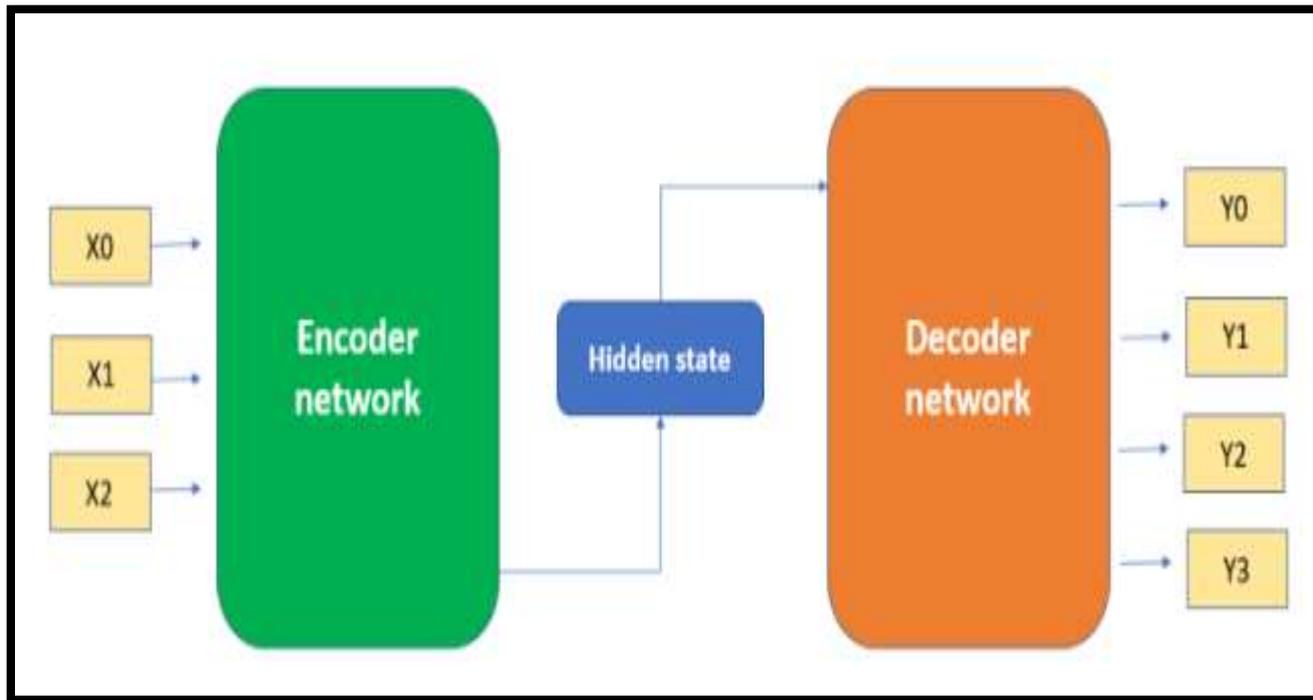
- 2. Decoder: The decoder takes the context vector produced by the encoder and generates an output sequence. It can be another sequence of different length, such as a translated sentence or a response in a chatbot.
- Like the encoder, the decoder often utilizes an RNN architecture. It takes the context vector as the initial hidden state and generates each element of the output sequence step by step.



- Let's take **machine translation** from **English to French** as an example.
- Given an **input sequence in English**: “They”, “are”, “watching”, “.”, this **encoder-decoder architecture** first **encodes the variable-length input into a state**, then **decodes the state** to generate the translated sequence, **token by token**, as output: “Ils”, “regardent”, “.”.



- The following picture represents the **encoder-decoder architecture** as explained here. Note that **both input and output sequences of data can be of varying length** as shown in the picture below.



SEQ TO SEQ MODEL

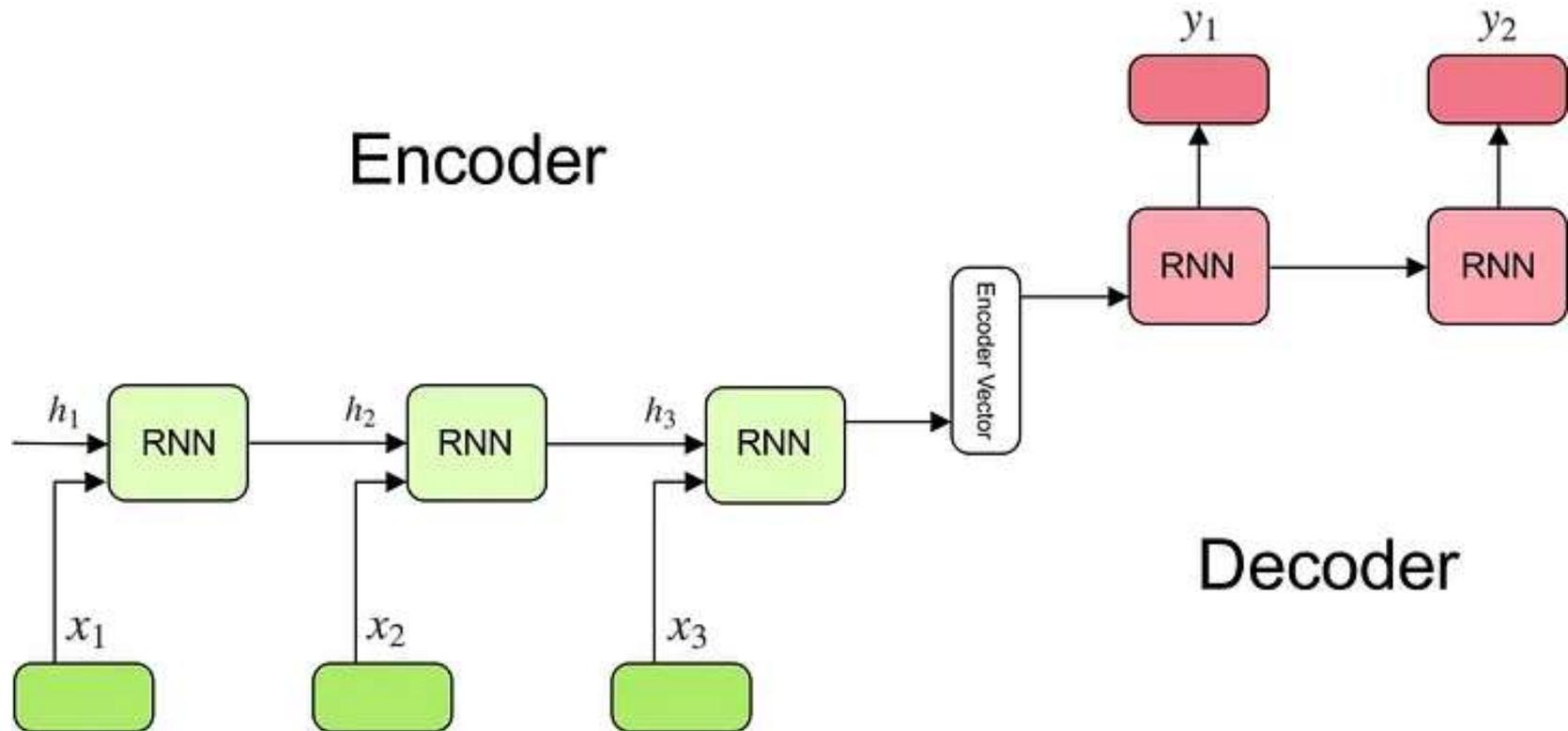
- Introduced for the first time in 2014 by Google, a sequence to sequence model aims to map a fixed-length input with a fixed-length output where **the length of the input and output may differ**.
- For example, translating “**What are you doing today?**” from English to Chinese has **input of 5 words and output of 7 symbols** (今天你在做什麼?). Clearly, we can't use a regular RNN network to map each word from the English sentence to the Chinese sentence.
- Sequence-to-sequence neural networks process a **variable length sequence of input data** (a text sentence, a time series, radio signals, etc.) and produce in output another data sequence, in general of a different length (e.g., a translation of the input sentence) .

- Most sequence-to-sequence NNs are based on the encoder-decoder (Enc/Dec) architecture and produce in output another data sequence, in general of a different length (e.g., a translation of the input sentence) .
- Most sequence-to-sequence NNs are based on the encoder-decoder (Enc/Dec) architecture



HOW SEQUENCE MODEL WORKS

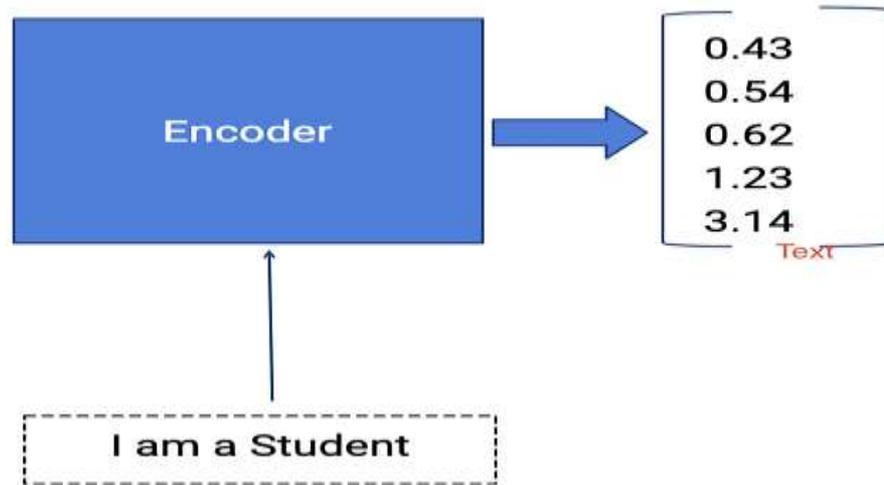
- In order to fully understand the model's underlying logic, we will go over the below illustration:



- Encoder: Multiple RNN cells can be stacked together to form the encoder. RNN reads each inputs sequentially
 - For every timestep (each input) t , the hidden state (hidden vector) h is updated according to the input at that timestep $X[i]$.
 - After all the inputs are read by encoder model, the final hidden state of the model represents the context/summary of the whole input sequence.
 - Example: Consider the input sequence “I am a Student” to be encoded. There will be totally 4 timesteps (4 tokens) for the Encoder model.
- 

➤ At each time step, the **hidden state h** will be **updated using the previous hidden state** and the current input.

Example: Encoder



At the **first timestep t_1** , the **previous hidden state h_0** will be considered as zero or randomly chosen. So the first RNN cell will update the current hidden state with the first input and h_0 .

- Each layer outputs two things – updated hidden state and the output for each stage. The outputs at each stage are rejected and only the hidden states will be propagated to the next layer.
- The hidden states h_i are computed using the formula:

$$h_t = f(W^{(hh)}h_{t-1} + W^{(hx)}x_t)$$

where,

h_t = hidden state

h_{t-1} = previous hidden state

$W^{(hh)}$ = weights attached to the previous hidden state. (h_{t-1})

x_t = input vector

$W^{(hx)}$ = weights attached to the input vector.



- At **second timestep t_2** , the **hidden state h_1** and the **second input $X[2]$** will be given as input, and the **hidden state h_2** will be updated according to both inputs.
- This simple formula represents the result of an ordinary recurrent neural network. As you can see, we just apply the appropriate weights to the **previously hidden state $h_{(t-1)}$** and the **input vector x_t** .

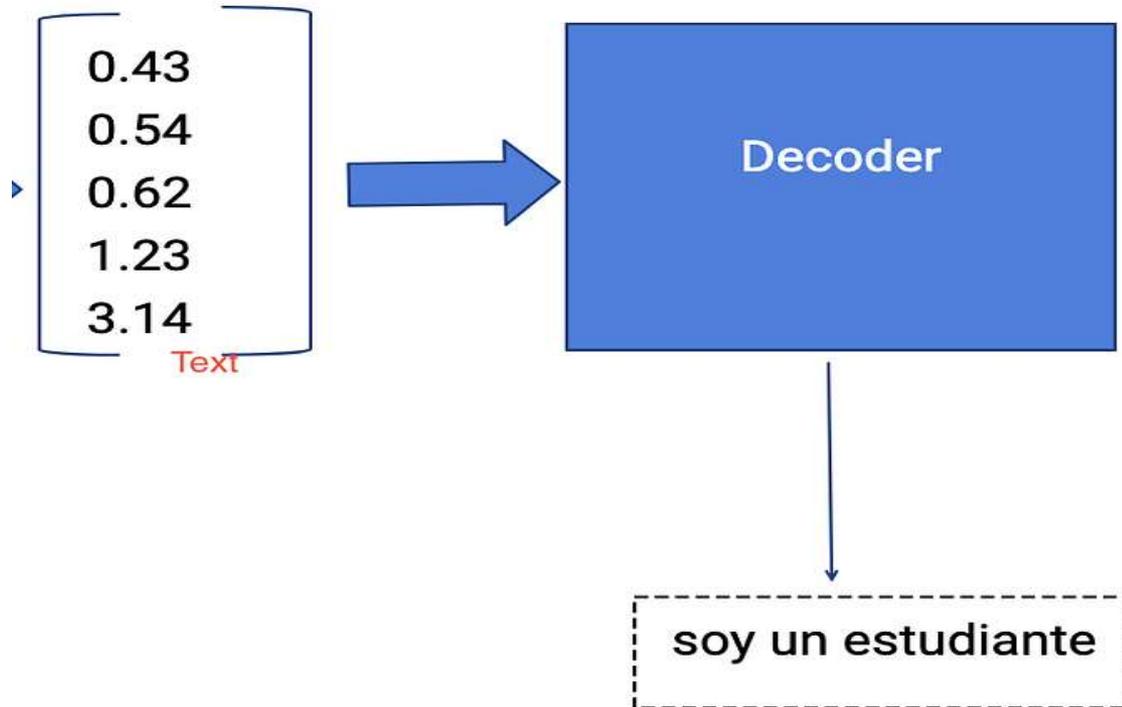


- 2. Encoder Vector: This is the **final hidden state** produced from the **encoder part** of the model. It is calculated using the formula above.
- This vector aims to encapsulate the information for all input elements in order to help the **decoder make accurate predictions**.
- It acts as the **initial hidden state of the decoder part of the model**.



- 3. Decoder : The Decoder generates the output sequence by predicting the next output Y_t given the hidden state h_t .
- The input for the decoder is the final hidden vector obtained at the end of encoder model.
- Each layer will have three inputs, hidden vector from previous layer h_{t-1} and the previous layer output y_{t-1} , original hidden vector h .
- t the first layer, the output vector of encoder and the random symbol **START**, empty hidden state h_{t-1} will be given as input, the outputs obtained will be y_1 and updated hidden state h_1 .





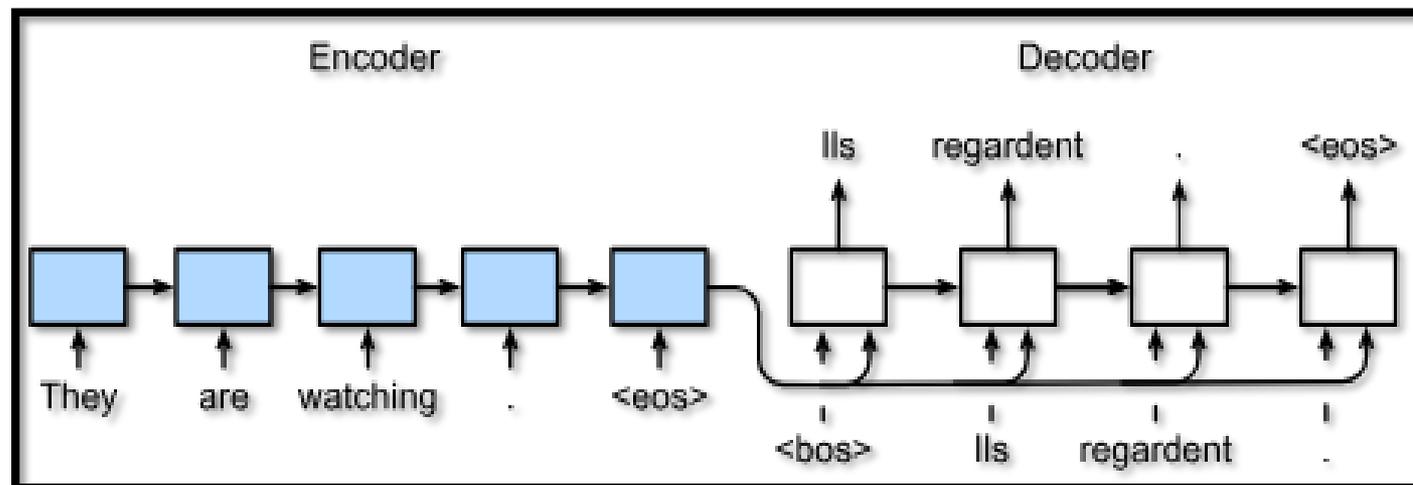
Example: Decoder



- Any hidden state h_i is computed using the formula:

$$h_t = f(W^{(hh)} h_{t-1})$$

- As you can see, we are just using the **previous hidden state to compute the next one.**
- Example:**



ADVANTAGES

- 1. Flexibility with Input and Output Sequences: Seq2Seq models can handle **variable-length input and output sequences**, particularly those using the **encoder-decoder architecture**. This makes them suitable for tasks like machine translation, where the length of the input sentence (e.g., English) and the output sequence (e.g., French) can differ significantly.
- 2. Versatility in Application: Seq2Seq models are not limited to text-based tasks. They are also employed in **speech recognition, video captioning, and time series prediction.**

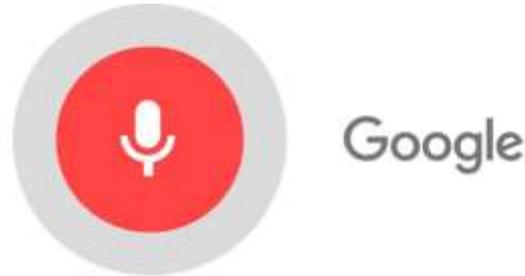


APPLICATIONS

- A sequence to sequence model lies behind numerous systems which you face on a daily basis. For instance, seq2seq model powers applications like **Google Translate, voice-enabled devices and online chatbots**. Generally speaking, these applications are composed of:
 - **1. Machine Translation:** a 2016 paper from Google shows how the seq2seq model's translation quality “approaches or surpasses all currently published results”.



- **2. *Speech recognition*** – another [Google paper](#) compares the existing seq2seq models on the speech recognition task.



- **3. *Video captioning*** – a 2015 [paper](#) shows how a seq2seq yields great results on generating movie descriptions.



S2VT: A herd of zebras are walking in a field.



UNIT-IV

Long Short Term Memory

CONTENTS

- ❖ Disadvantages of RNN
- ❖ Introduction to LSTM
- ❖ Architecture of LSTM
- ❖ How LSTM Works
- ❖ Difference between RNN and LSTM
- ❖ Advantages
- ❖ Step by Step Explanation

DRAWBACKS OF RNN

- Drawbacks of RNN: An RNN can only remember the **immediate past input**. It can't use inputs from several previous sequences to improve its prediction.
- 1. The **problem with Recurrent Neural Networks** is that they **simply store the previous data** in their “short-term memory”. Once the memory in it runs out, it simply deletes the longest retained information and **replaces it with new data**.
- 2. Vanishing Gradient Problem: The **Vanishing Gradient Problem** is a **drawback of recurrent neural networks (RNNs)** that occurs when **gradients used to update the network's weights** become **very small** as they are **propagated back through many time steps**.

- This can make it **difficult for RNNs to learn large data sequences.**
- 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.**
- Based on the above Two Problems we can use **LSTM.**



LONG SHORT TERM MEMORY

- **Long Short-Term Memory (LSTM)** is a type of **artificial recurrent neural network (RNN) architecture** used in the field of deep learning.
- Long Short-Term Memory is an **improved version of recurrent neural network** designed by **Hochreiter & Schmidhuber** in the **Year 1991**.
- A **sentence or phrase** only holds meaning when **every word** in it is associated with its **previous word and the next one**.



- LSTM, is opposed to RNN, extends it by creating both short-term and long-term memory components to efficiently study and learn sequential data.
 - Hence, it's great for Machine Translation, Speech Recognition, time-series analysis, Audio and Video Classification etc.
 - Long Short Term Memory (LSTM) networks are a powerful variant of Recurrent Neural Networks (RNNs) designed to handle long-term dependencies in sequential data. The core architectural advantage of LSTMs over traditional RNNs lies in their memory cells and gating mechanisms.
- 

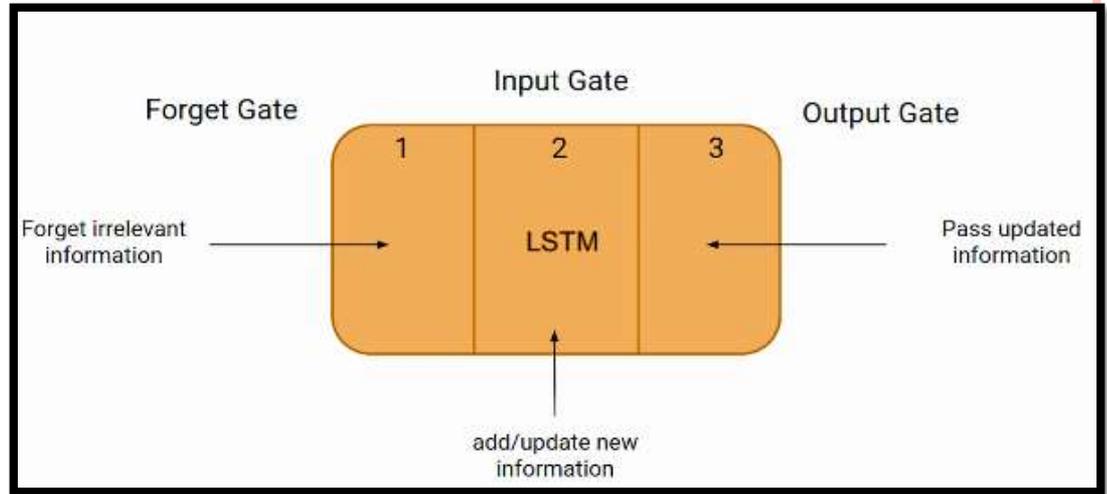
- Unlike RNNs, which struggle with the vanishing gradient problem, LSTMs incorporate specialized gates that manage the flow of information.
- LSTMs model address this problem by introducing a memory cell, which is a container that can hold information for an extended period.



LSTM ARCHITECTURE

➤ The LSTM architecture involves the **memory cell** which is controlled by **three gates**:

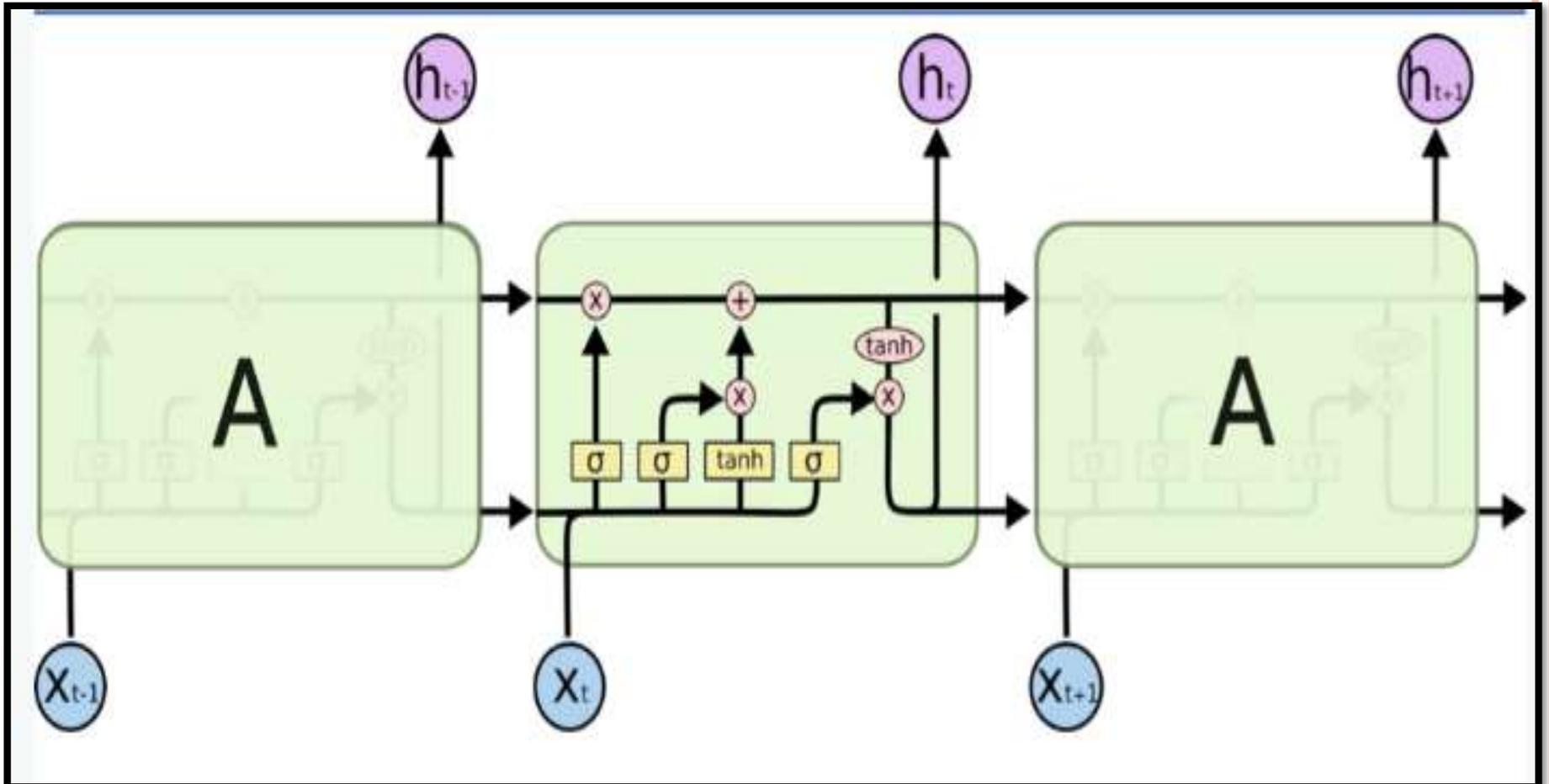
- **The input gate,**
- **The forget gate,** and
- **The output gate.**

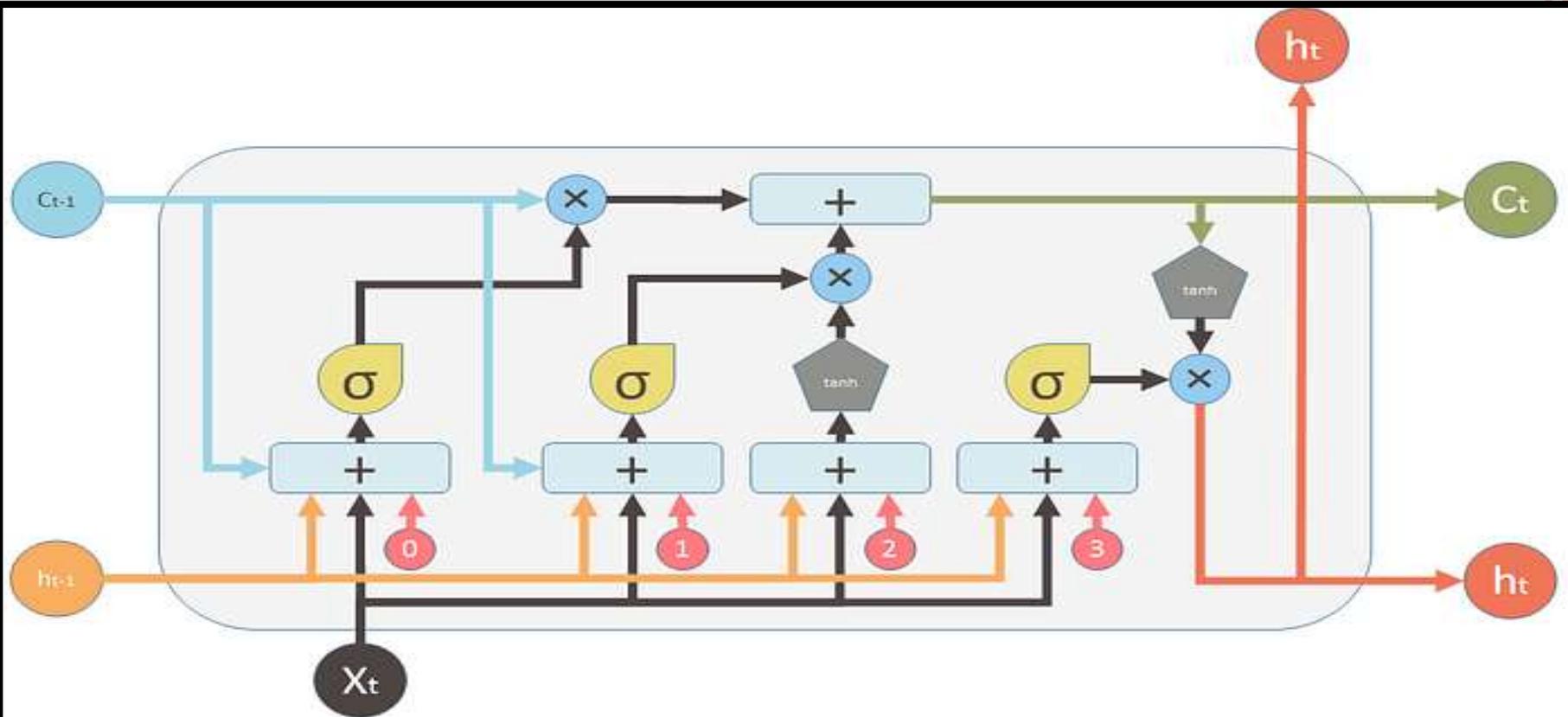


- These gates decide what **information to add to, remove from,** and **output from the memory cell.**
- **The input gate controls** what information is **added to the memory cell.**



- ❖ The forget gate controls what information is removed from the memory cell.
- ❖ The output gate controls what information is output from the memory cell.
- ❖ Cell (the memory part of LSTM): The cell stores the state of a sequence, so it has the ability to either keep or forget certain information.
- ❖ The LSTM maintains a hidden state, which acts as the short-term memory of the network. The hidden state is updated based on the input, the previous hidden state, and the memory cell's current state.





Inputs:

- X_t Input vector
- c_{t-1} Memory from previous block
- h_{t-1} Output of previous block

outputs:

- c_t Memory from current block
- h_t Output of current block

Nonlinearities:

- σ Sigmoid
- tanh Hyperbolic tangent

Bias: 0

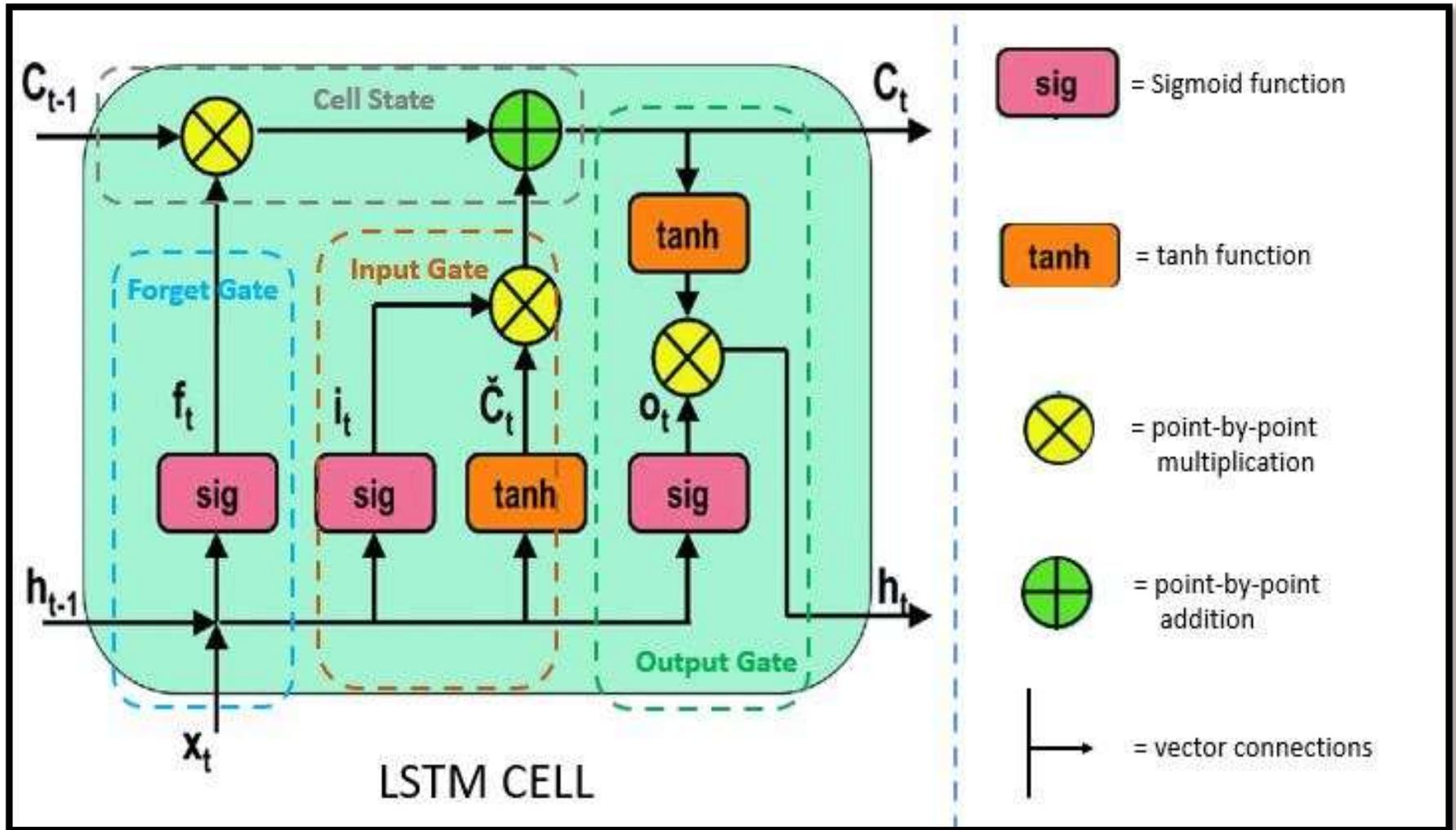
Vector operations:

- \otimes Element-wise multiplication
- $+$ Element-wise Summation / Concatenation

HOW DOES IT WORKS?

- ❖ Here **Information** is retained by the cells and the memory manipulations are done by the gates.
- ❖ Let's assume we have a **sequence of words** ($w_1, w_2, w_3, \dots, w_n$) and we are **processing the sequence one word at a time**. Let's denote the **state of the LSTM at time step t** as (h_t, c_t),
 - ❖ where h_t is the **hidden state** and c_t is the **cell state**.
- c_{t-1} stands for the **input from a memory cell in time point t** ;
 X_t is an **input in time point t** ;
- h_t is an **output in time point t** that **goes to both the output layer** and the **hidden layer in the next time point**.

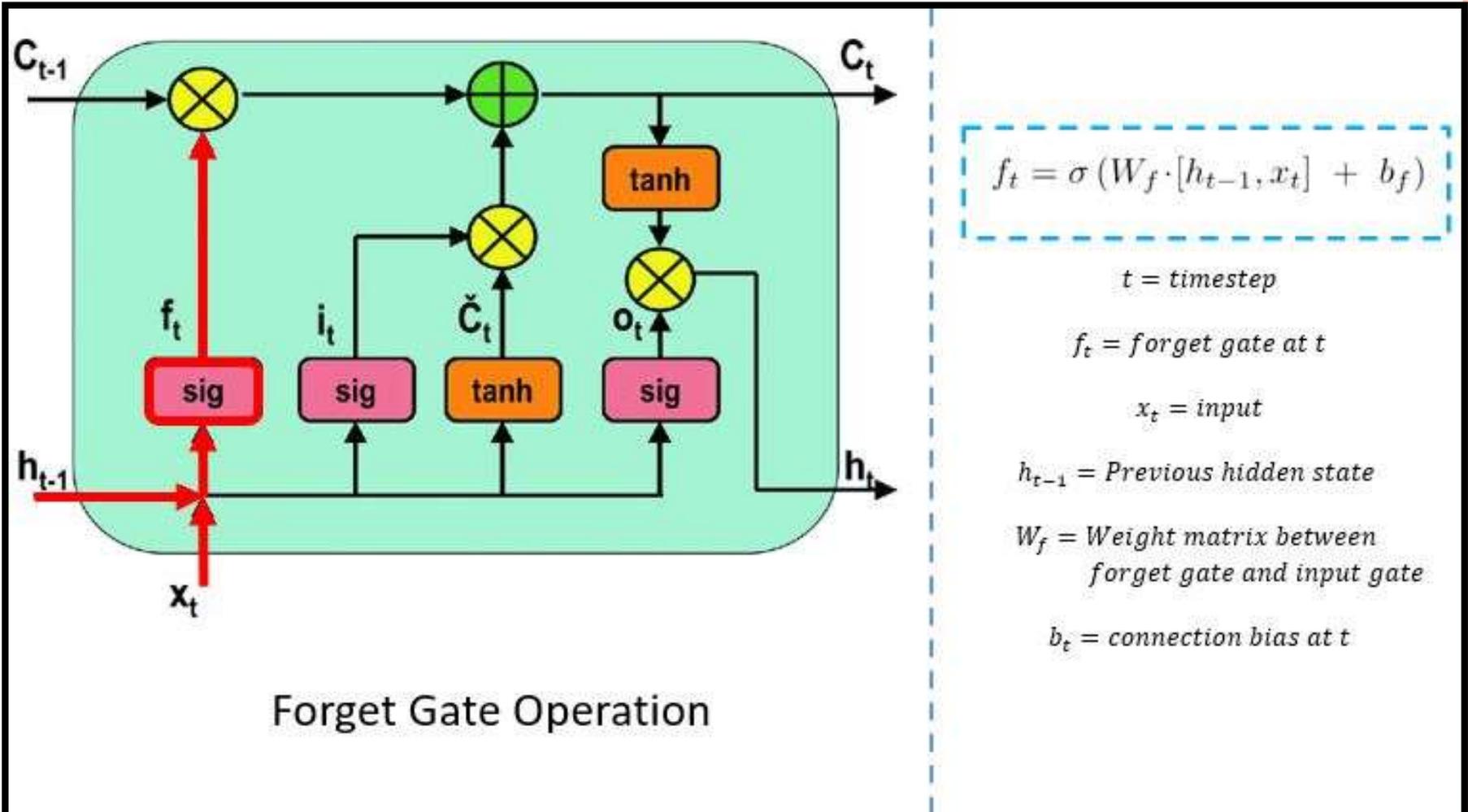




- ❖ Thus, every block has three inputs (x_t , h_{t-1} , and c_{t-1}) and two outputs (h_t and c_t). An important thing to remember is that all these inputs and outputs are not single values, but vectors with lots of values behind each of them.
- ❖ Step-1: The LSTM receives the input vector (x_t) and the previous state (h_{t-1} , c_{t-1}).
- ❖ Step-2: Forget State: The information that is no longer useful in the cell state is removed with the forget gate. Two inputs x_t (input at the particular time) and h_{t-1} (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**.
- ❖ Role: By **selectively forgetting irrelevant data**, the **forget gate** helps in **preventing the cell state from being cluttered with unnecessary information**.
- ❖ The equation for the forget gate is:





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

$t = \text{timestep}$

$f_t = \text{forget gate at } t$

$x_t = \text{input}$

$h_{t-1} = \text{Previous hidden state}$

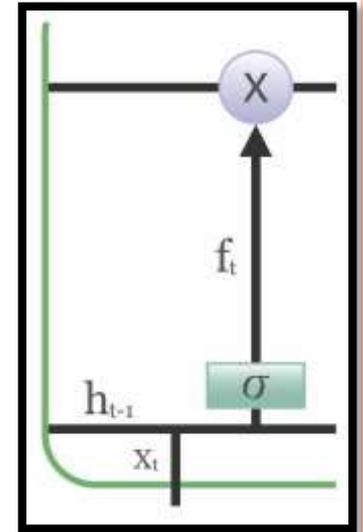
$W_f = \text{Weight matrix between forget gate and input gate}$

$b_t = \text{connection bias at } t$

[$h_{t-1} + x_t$] given to sigmoid and the value is stored in f_t



$$f_t = \sigma \left(W_f \cdot [h_{(t-1)}, x_t] + b_f \right)$$



➤ 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**.



- ❖ Step-3: Input Gate: The **input gate (it)** decides what new information to store in the cell state.
- ❖ It has two parts. A **sigmoid Function** called the "**input gate layer**" decides **which values we'll update**, and a **tanh Function** creates a **vector of new candidate values (C_t^{\sim})** that could be added to the state. Ie, 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.**

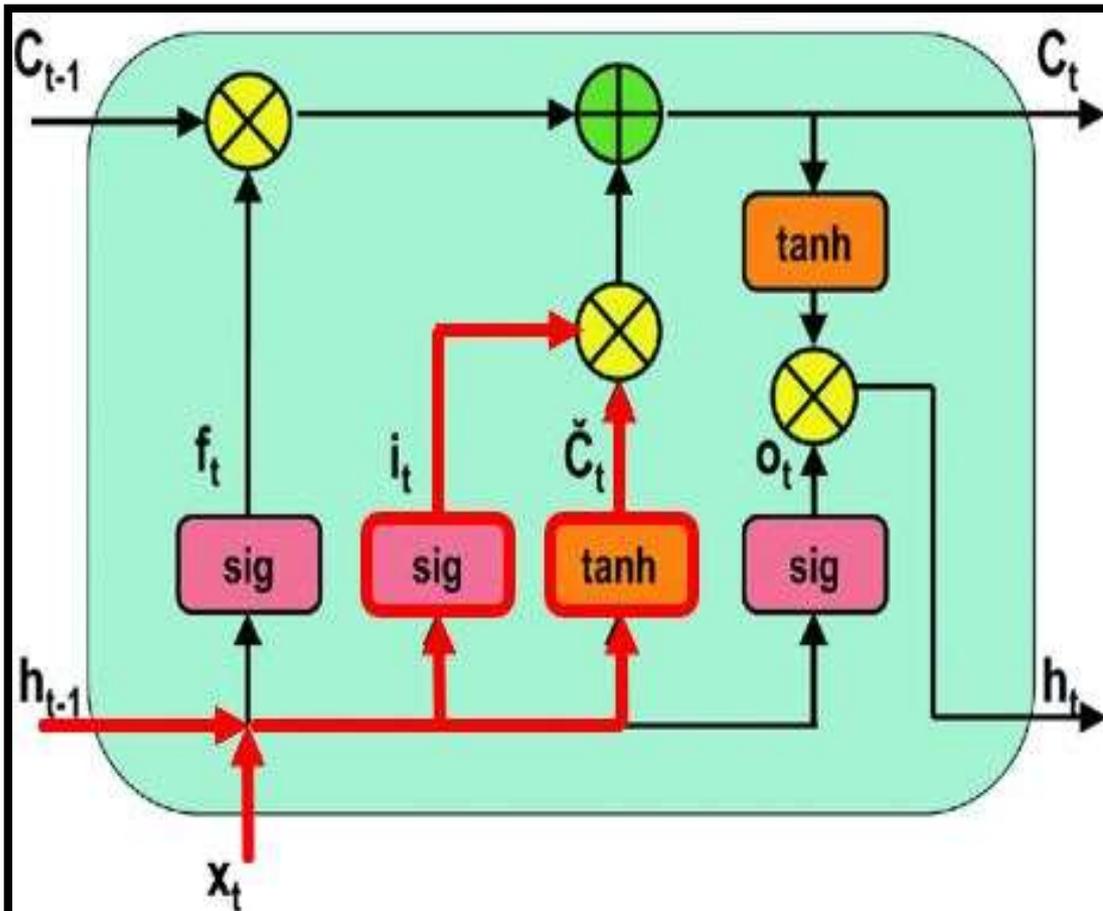
- ❖ The equation for the input gate is:

$$i_t = \sigma \left(W_i \cdot [h_{(t-1)}, x_t] + b_i \right)$$

- ❖ Candidate Values(Cell State Update):

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





Input Gate Operation

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

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

$t = \text{timestep}$

$i_t = \text{input gate at } t$

$W_i = \text{Weight matrix of sigmoid operator between input gate and output gate}$

$b_t = \text{bias vector at } t$

$\tilde{C}_t = \text{value generated by tanh}$

$W_C = \text{Weight matrix of tanh operator between cell state information and network output}$

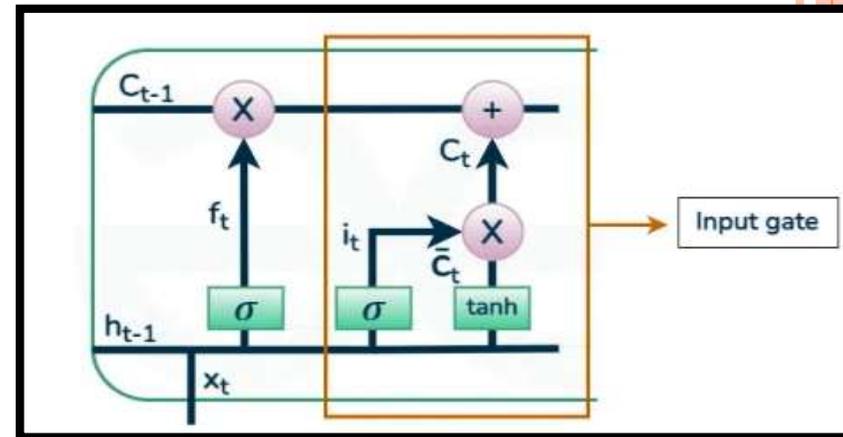
$b_C = \text{bias vector at } t, \text{ w.r.t } W_C$

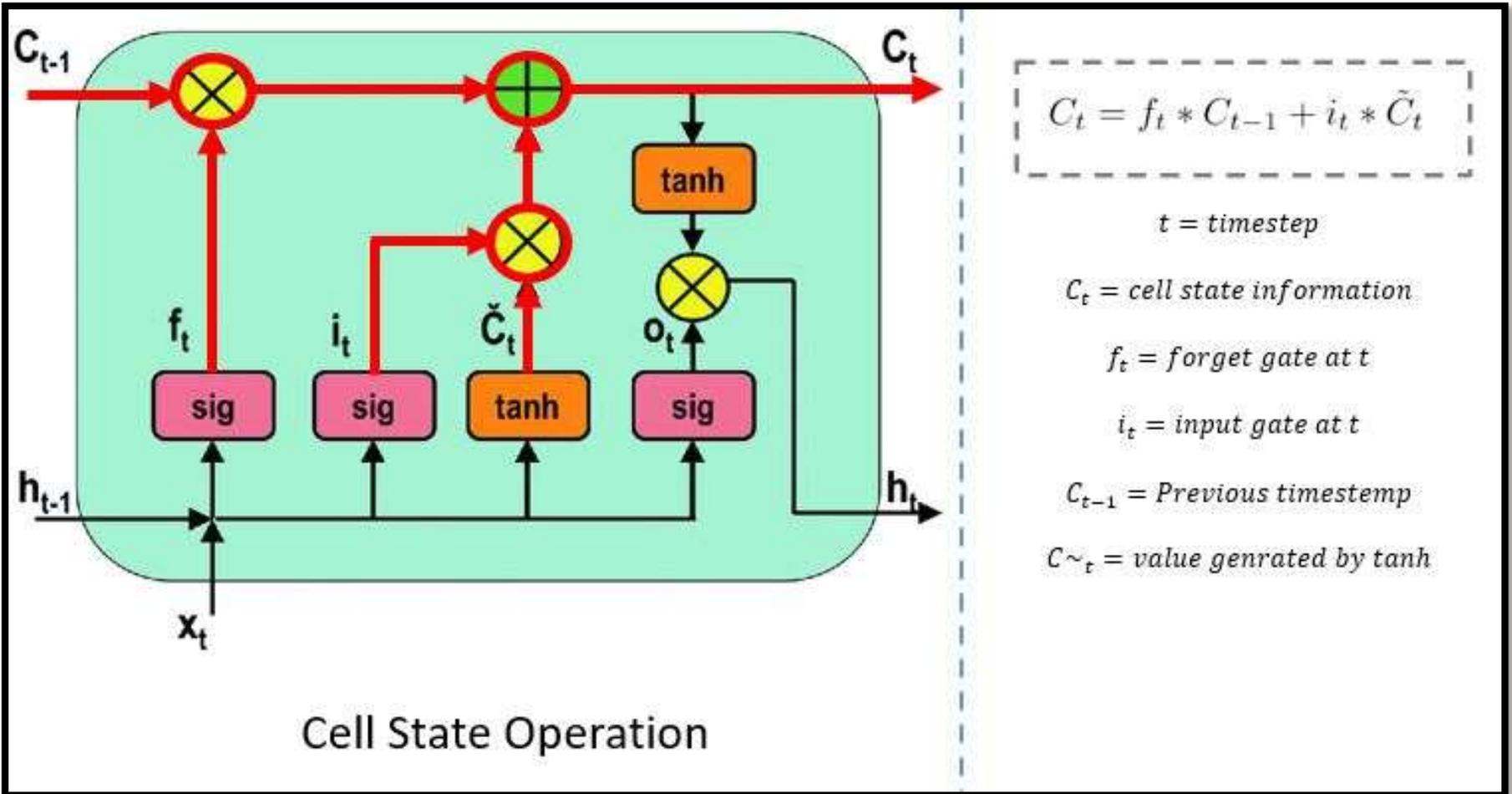
- ❖ We multiply the previous state by f_t , disregarding the information we had previously chosen to ignore. Next, we include $i_t * C_t$. This represents the updated candidate values, adjusted for the amount that we chose to update each state value.

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

where

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





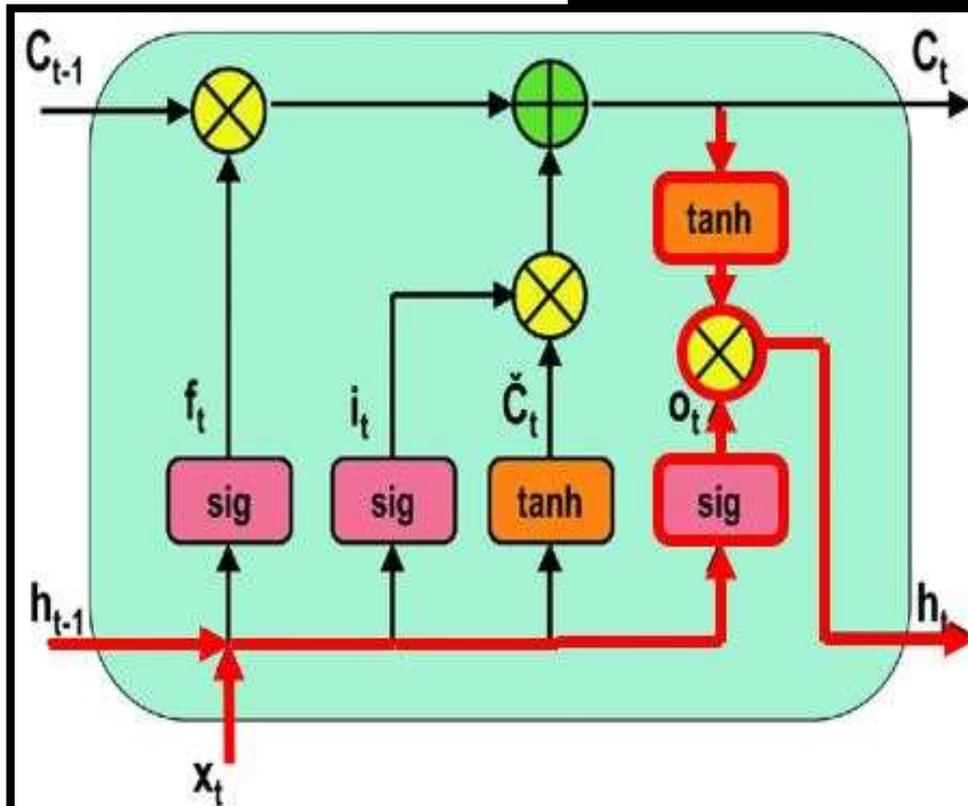
- ❖ Step-4: Output Gate: The task of extracting useful information from the **current cell state** to be **presented as output is done by the output gate**. First, a **vector is generated** by **applying tanh function** on the cell. Then, the information is regulated **using the sigmoid function**. At 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:

$$o_t = \sigma \left(W_o \cdot [h_{(t-1)}, x_t] + b_o \right)$$

❖ Hidden State:

$$h_t = o_t \times \tanh(C_t)$$



Output Gate Operation

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

$$h_t = o_t * \tanh(C_t)$$

$t = \text{timestep}$

$O_t = \text{output gate at } t$

$W_o = \text{Weight matrix of output gate}$

$b_o = \text{bias vector, w.r.t } W_o$

$h_t = \text{LSTM output}$

LSTM Vs RNN

| Feature | LSTM (Long Short-term Memory) | RNN (Recurrent Neural Network) |
|----------------------------------|---|--|
| Memory | Has a special memory unit that allows it to learn long-term dependencies in sequential data | Does not have a memory unit |
| Directionality | Can be trained to process sequential data in both forward and backward directions | Can only be trained to process sequential data in one direction |
| Training | More difficult to train than RNN due to the complexity of the gates and memory unit | Easier to train than LSTM |
| Long-term dependency learning | Yes | Limited |
| Ability to learn sequential data | Yes | Yes |
| Applications | Machine translation, speech recognition, text summarization, natural language processing, time series forecasting | Natural language processing, machine translation, speech recognition, image processing, video processing |

ADVANTAGES

- 1. Ability to process sequential data: LSTMs are designed to work with **sequential data**, such as **time series data** or **natural language text**. This makes them well-suited for a wide range of applications, including **speech recognition**, **language translation**, and **sentiment analysis**.
- 2. Ability to handle long-term dependencies: LSTMs are specifically designed to **address the problem of vanishing gradients**, which can occur in **traditional RNNs** when trying to **process long sequences**. This makes them well-suited for tasks that require processing **long-term dependencies**, such as **predicting stock prices** or **weather patterns**.

- **3. Memory cell:** The memory cell in an LSTM allows the network to selectively remember or **forget information** over long periods of time, making it **more effective at handling complex tasks** than other types of RNNs.



$f_t = \sigma(W_f S_{t-1} + W_f X_t)$ - Forget Gate

$i_t = \sigma(W_i S_{t-1} + W_i X_t)$ - Input Gate

$o_t = \sigma(W_o S_{t-1} + W_o X_t)$ - Output Gate

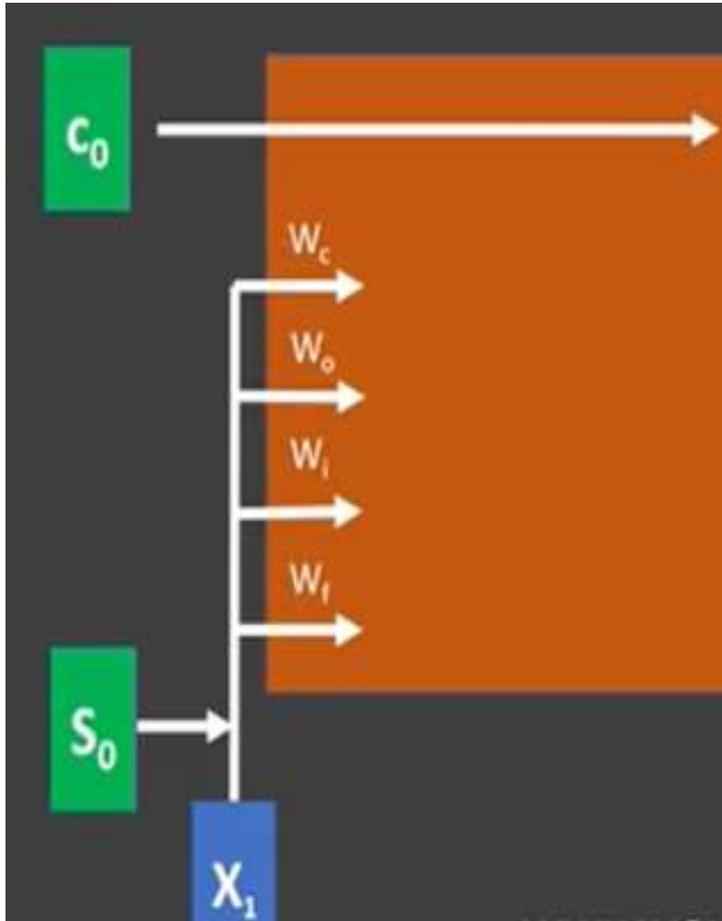
$\tilde{C}_t = \tanh(W_c S_{t-1} + W_c X_t)$

$c_t = (i_t * \tilde{C}_t) + (f_t * c_{t-1})$ - Cell State

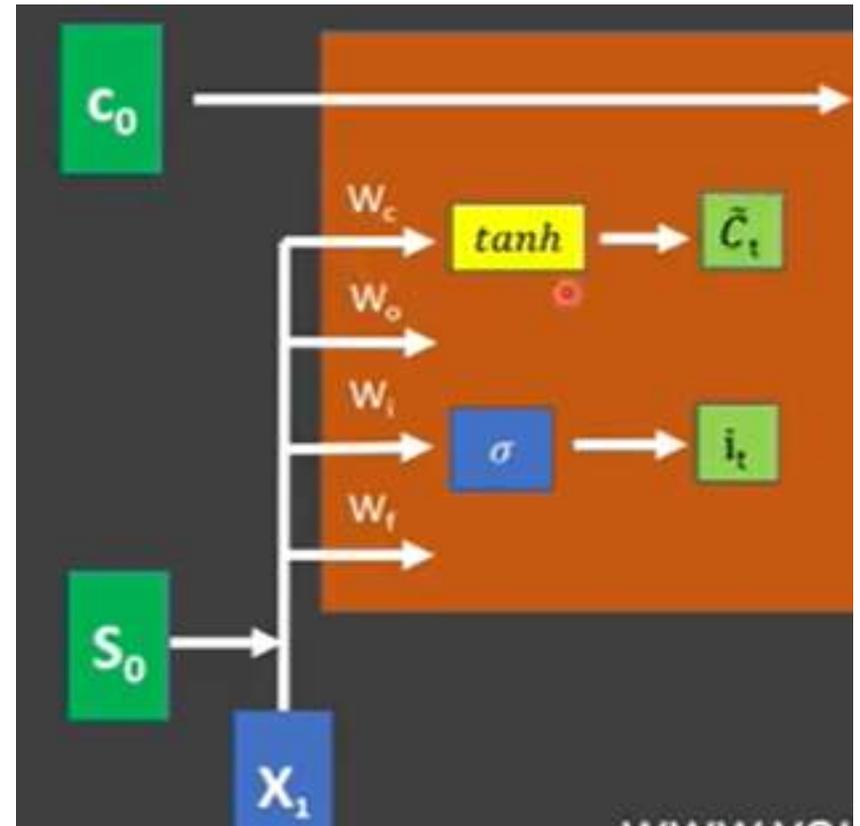
$h_t = o_t * \tanh(c_t)$ - New State



Step-1:



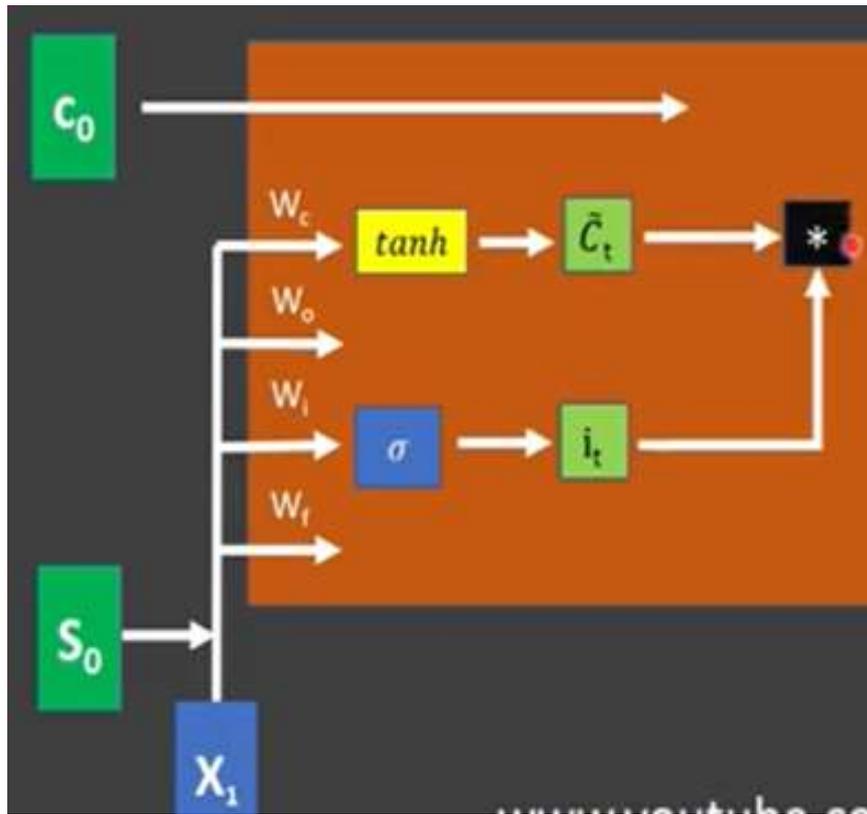
Step-2:



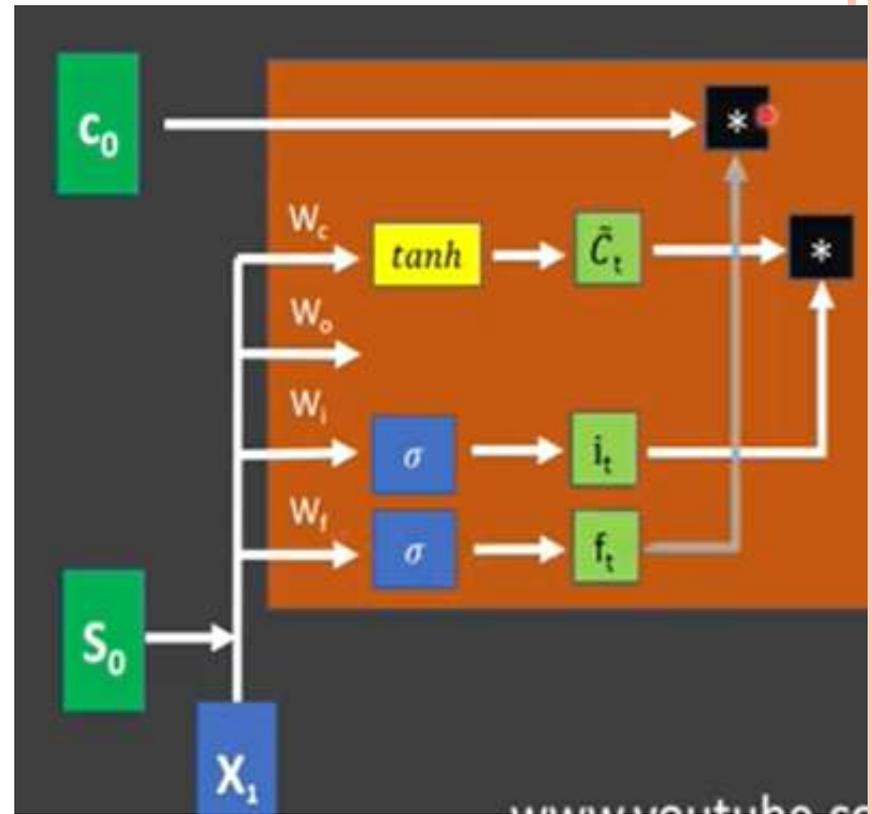
$$i_t = \sigma(W_i S_{t-1} + W_i X_t) \quad \text{- Input Gate}$$

$$\tilde{C}_t = \tanh(W_c S_{t-1} + W_c X_t)$$

Step-3:

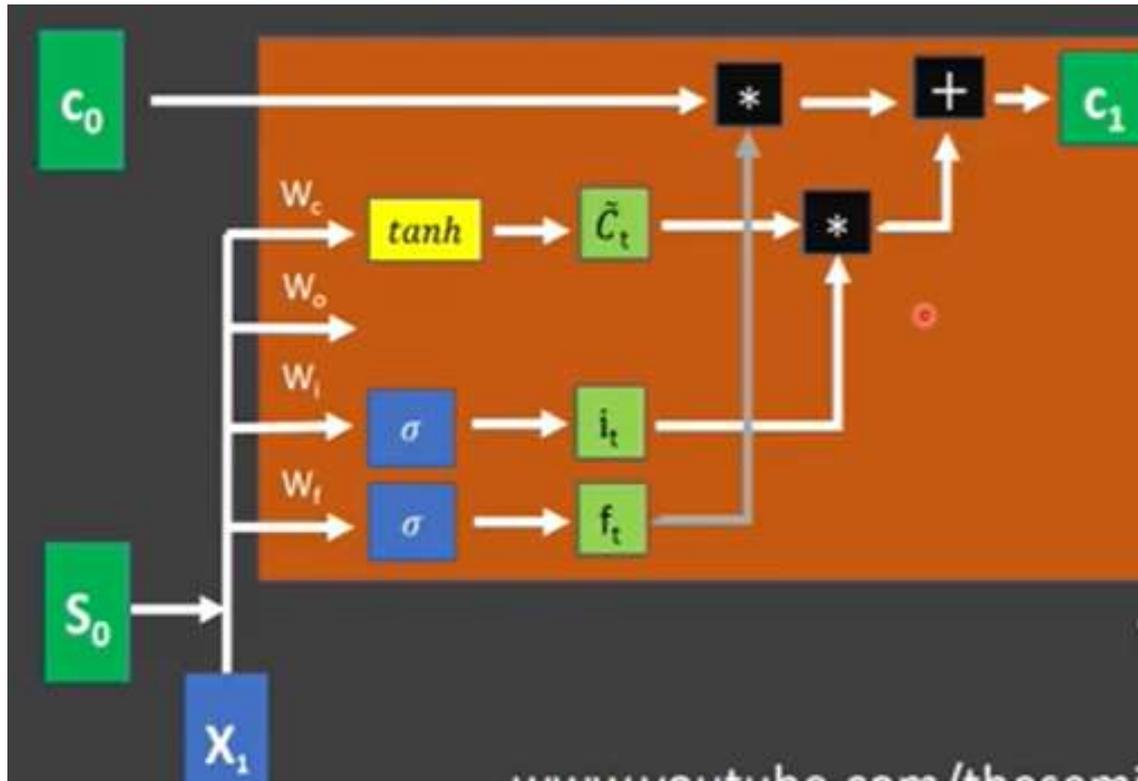


Step-4:



$$f_t = \sigma(W_f S_{t-1} + W_f X_t) - \text{Forget Gate}$$

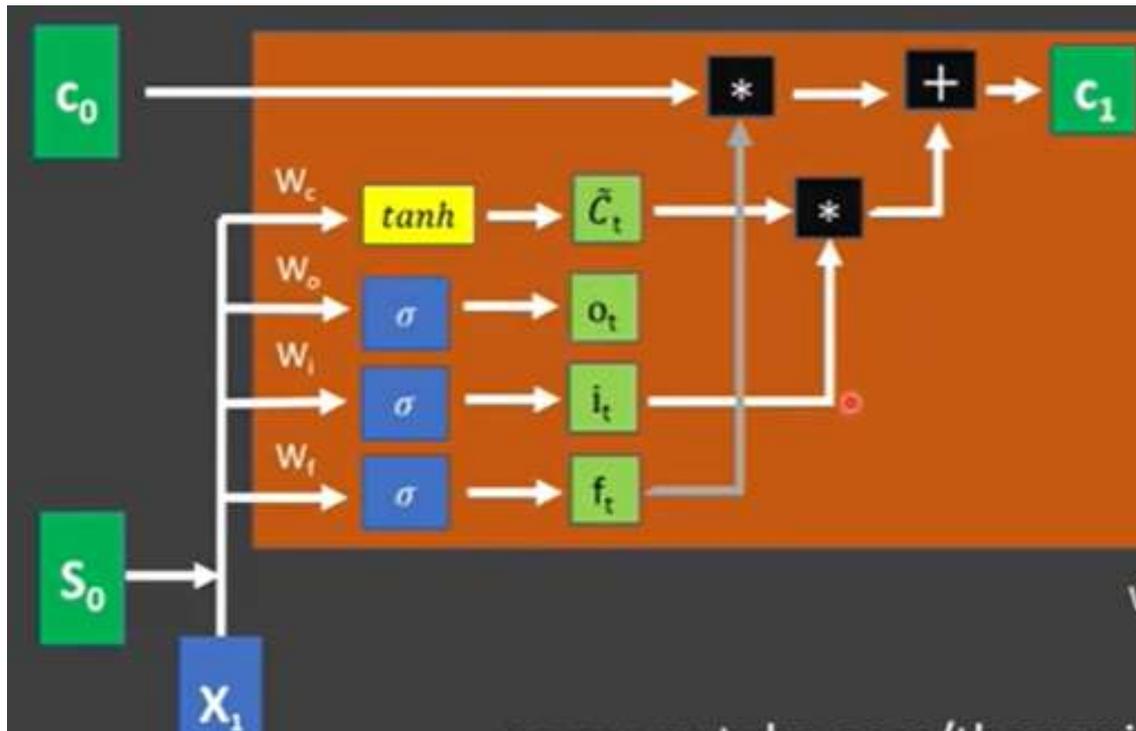
Step-5:



$$c_t = (i_t * \tilde{c}_t) + (f_t * c_{t-1}) - \text{Cell State}$$



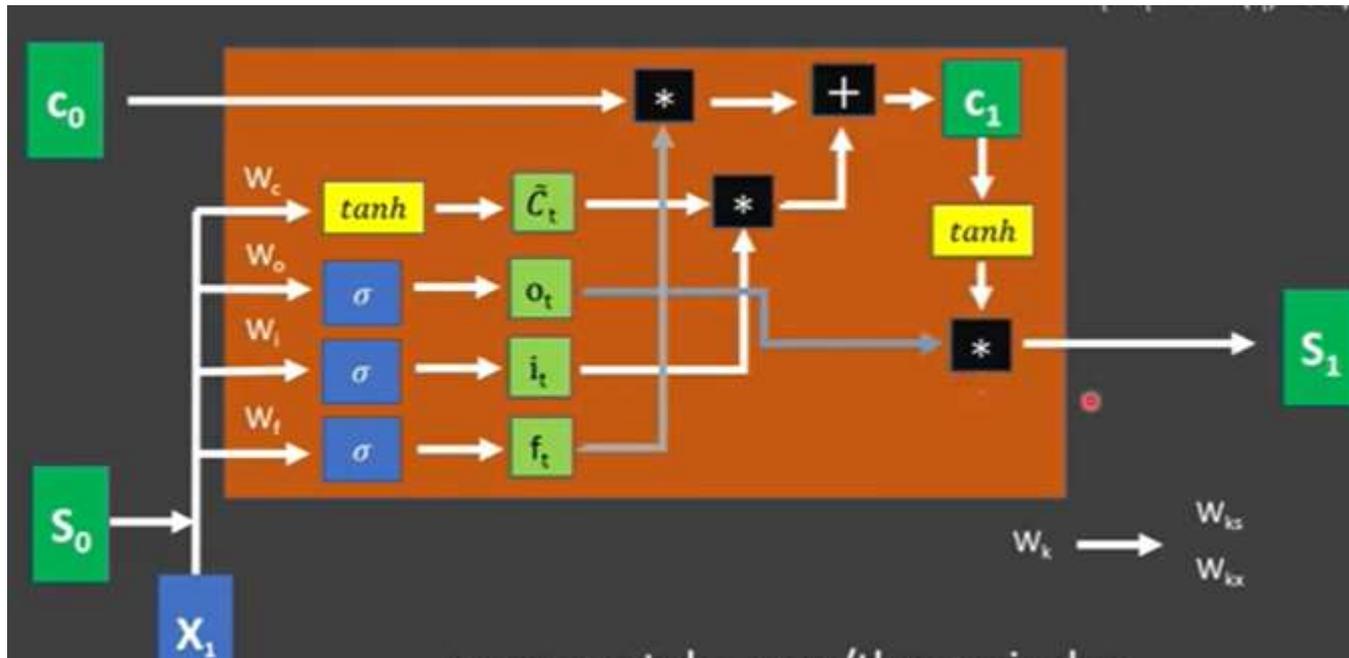
Step-6:



$$o_t = \sigma(W_o s_{t-1} + W_o x_t) - \text{Output Gate}$$



Step-7:



$$h_t = o_t \cdot \tanh(c_t) - \text{New State}$$

