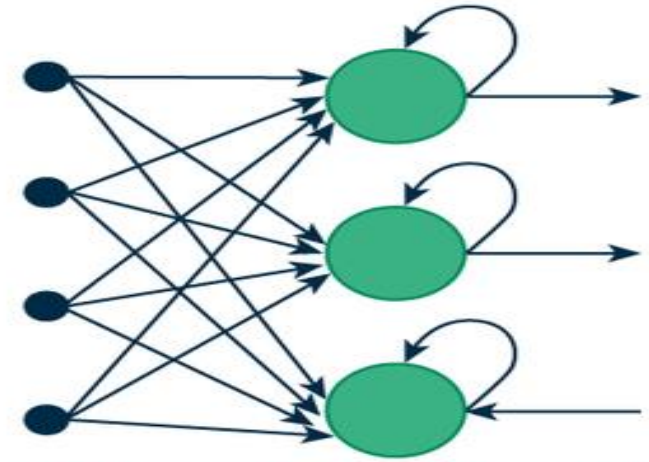
Recurrent Neural Network(RNN)

- Recurrent Neural Networks (RNNs) were introduced to address the limitations of traditional neural networks, such as Feed Forward Neural Networks (FNNs), when it comes to processing sequential data.
- FNN takes inputs and process each input independently through a number of hidden layers without considering the order and context of other inputs.
- Due to which it is unable to handle sequential data effectively and capture the dependencies between inputs.
- As a result, FNNs are not well-suited for sequential processing tasks such as, language modeling, machine translation, speech recognition, time series analysis, and many other applications that requires sequential processing.
- To address the limitations posed by traditional neural networks, RNN comes into the picture.

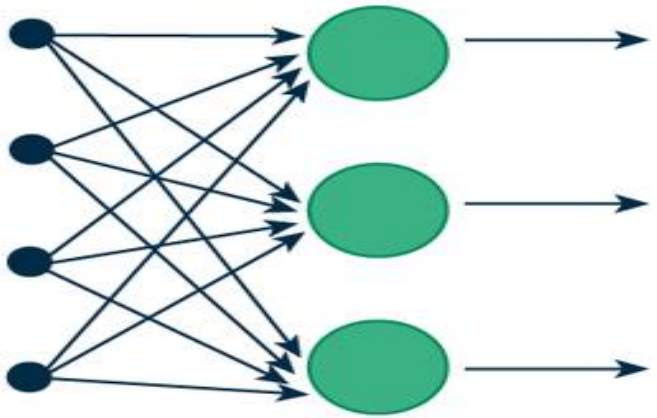
Recurrent Neural Network(RNN)

- RNN overcome these limitations by introducing a recurrent connection that allow information to flow from one time-step to the next.
- This recurrent connection enables RNNs to maintain internal memory, where the output of each step is fed back as an input to the next step.
- This allows the network to capture the information from previous steps and utilize it in the current step, enabling model to learn temporal dependencies and handle input of variable length.

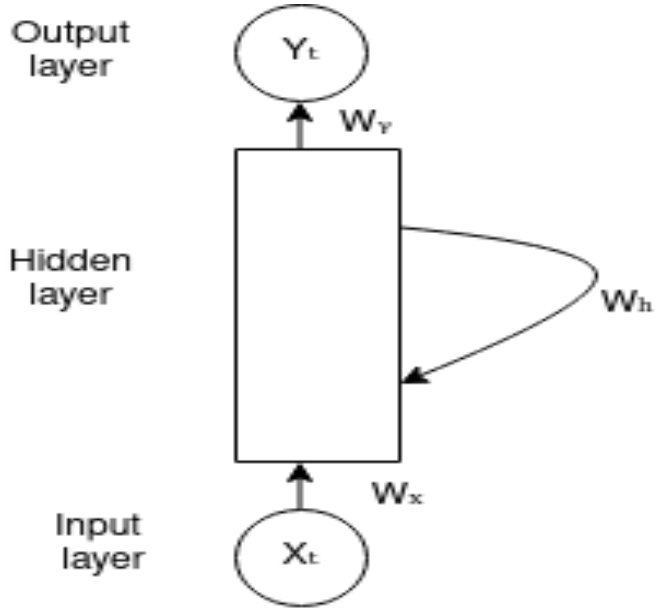
Recurrent Neural Network(RNN)

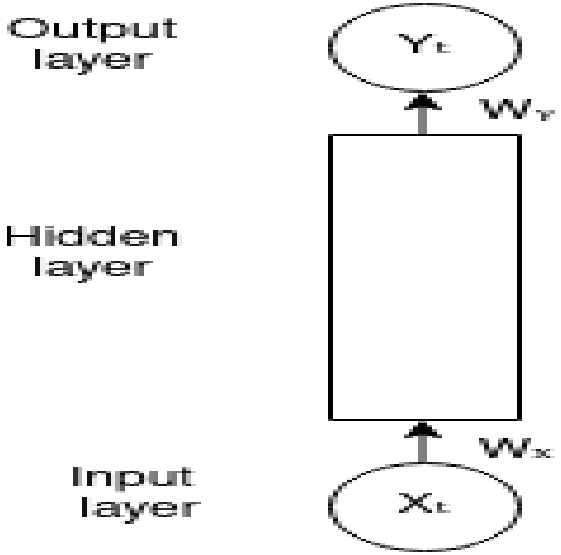


(a) Recurrent Neural Network



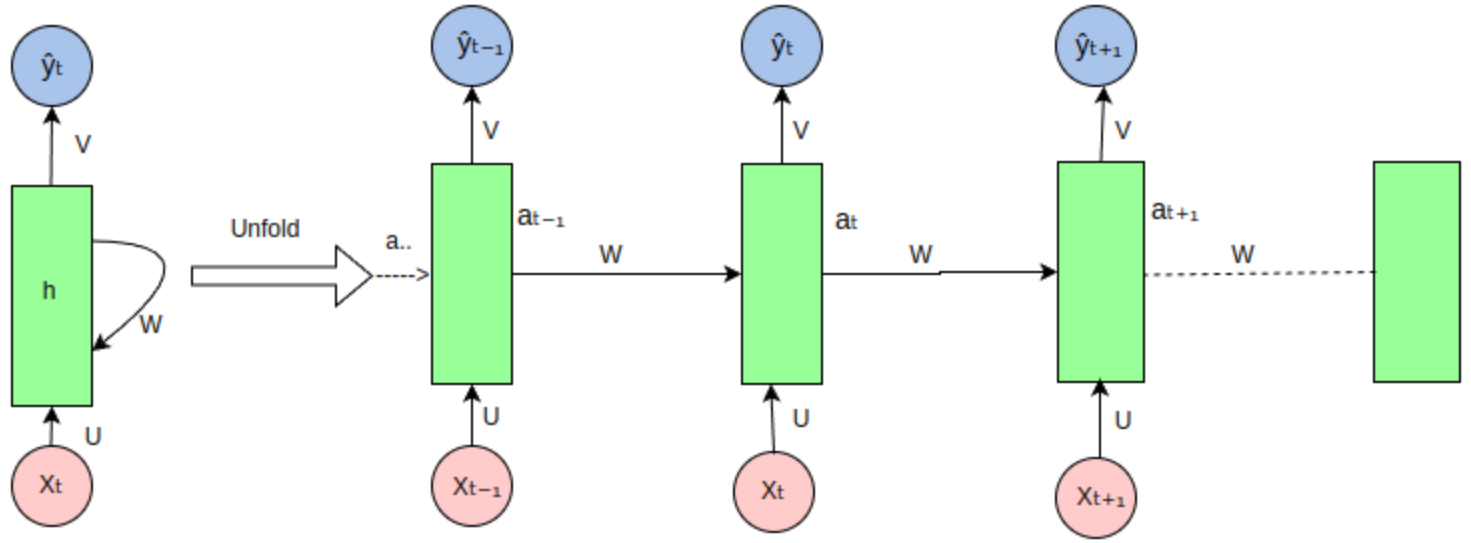
(b) Feed-Forward Neural Network





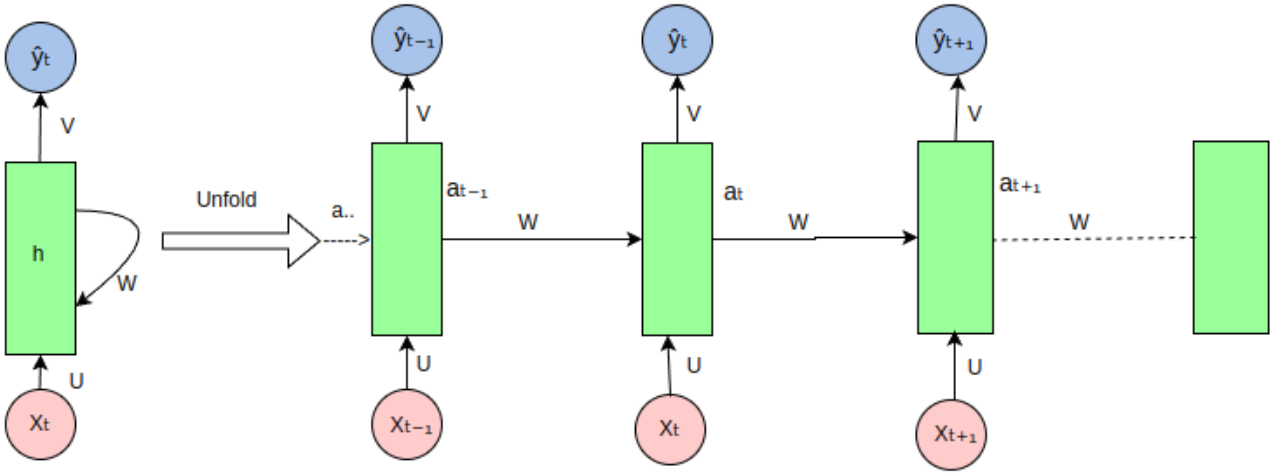
Architecture Of RNN

- The RNN takes an input vector **X** and the network generates an output vector **y** by scanning the data sequentially from left to right, with each time step updating the hidden state and producing an output.
- It shares the same parameters across all time steps. This means that, the same set of parameters, represented by U, V, W is used consistently throughout the network.



Architecture Of RNN

- U represents the weight parameter governing the connection from input layer **X** to the hidden layer **h**
- W represents the weight associated with the connection between hidden layers, and
- V for the connection from hidden layer **h** to output layer **y**.
- This sharing of parameters allows the RNN to effectively capture temporal dependencies and process sequential data more efficiently by retaining the information from previous input in its current hidden state.



Architecture Of RNN

- At each time step t, the hidden state a_t is computed based on the current input x_t , previous hidden state a_{t-1} and model parameters, θ.

$$a_t = f(a_{t-1}, x_t; \theta) \qquad (1)$$

- It can also be written as,

$$a_t = f(U * X_t + W * a_{t-1} + b) \qquad (2)$$

where,

- a_t represents the output generated from the hidden layer at time step t .
- x_t is the input at time step t.
- θ represents a set of learnable parameters(weights and biases).
- U, V, and W ∈ θ
- b is the bias vector for the hidden layer; b ∈ θ
- f is the activation function.

Architecture Of RNN

- For a finite number of time steps T=4, we can repeat it T-1 times.
- When T = 4, $a_4 = f(a_3, x_4; \theta)$ by eq.(1).
- Eq.(2) can be expanded as,
- $a_4 = f(U * X_4 + W * a_3 + b)$
- $a_3 = f(U * X_3 + W * a_2 + b)$
- $a_2 = f(U * X_2 + W * a_1 + b)$

Architecture Of RNN

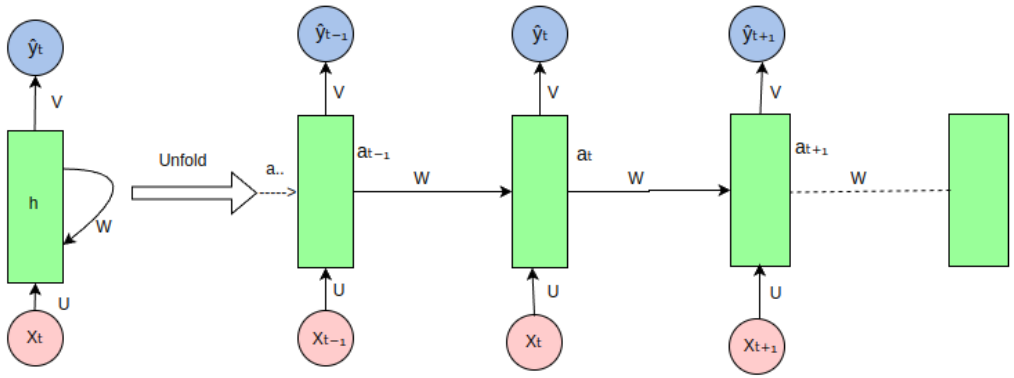
- The output at each time step t, denoted as , $\hat{y}_t$ is computed based on the hidden state output a_t using the following formula,

$$\hat{y}_t = f(a_t; \theta) \qquad (3)$$

- Eq.(3) can be written as,

$$\hat{y}_t = f(V * a_t + c) \qquad (4)$$

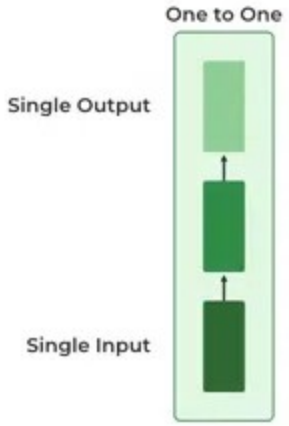
- when t=4, $\hat{y}_4 = f(V * a_4 + c)$
- where, $\hat{y}_t$ is the output predicted at time step t.
- c is the bias vector for the output layer.



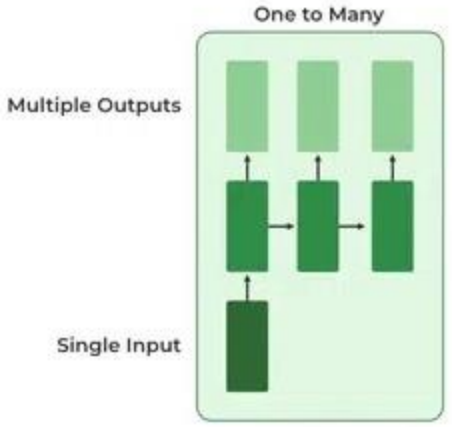
Variants of RNN Architectures

- RNNs can be adapted into different architectures based on the nature of the input and output:
- **One-to-One:**
  - A simple feedforward network, not typically considered an RNN, but included for completeness.
  - Example: Standard neural network for image classification.
- **One-to-Many:**
  - Used when a single input leads to a sequence of outputs.
  - Example: Generating a sequence of text from a single image (image captioning).
- **Many-to-One:**
  - Used for tasks like sentiment analysis where the entire sequence leads to a single output (e.g., classification).
  - Example: Predicting the sentiment of a sentence.
- **Many-to-Many:**
  - Used for tasks where each input in the sequence corresponds to an output (e.g., translation).
  - Example: Machine translation from one language to another.

- **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.

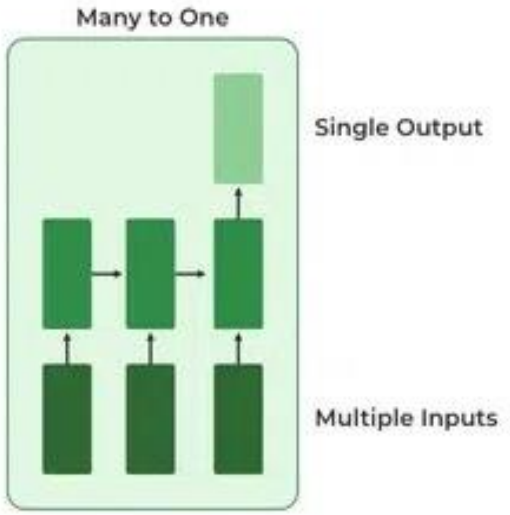


- **One To Many**
- In this type of RNN, there is one input and many outputs associated with it. One of the most used examples of this network is Image captioning where given an image we predict a sentence having Multiple words.



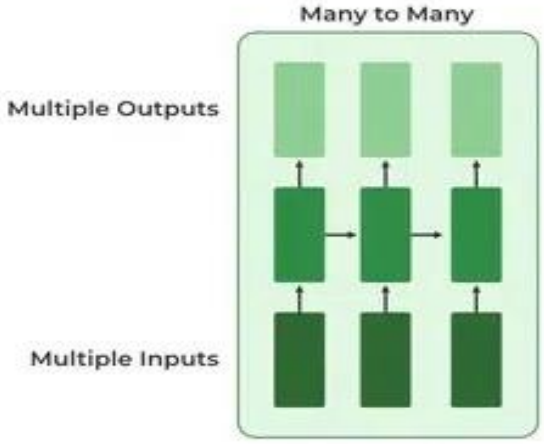
Many to One

- 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.



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.



Backpropagation Through Time (BPTT)

- Backpropagation involves adjusting the model's parameters (weights and biases) based on the error between predicted output and the actual target value.
- The goal of backpropagation is to improve the model's performance by minimize the loss function.
- Backpropagation Through Time is a special variant of backpropagation used to train RNNs, where the error is propagated backward through time until the initial time step t=1.
- Backpropagation involves two key steps: forward pass and backward pass.

Backpropagation Through Time (BPTT)

1. **Forward Pass**:

- During forward pass, the RNN processes the input sequence through time, from t=1 to t=n, where n is the length of input sequence.
- In each forward propagation, the following calculation takes place

$$a_t = U * X_t + W * a_{t-1} + b$$

$a_t = \sigma(a_t)$ [Generally, tanh() function is used]

$\hat{y}_t = \text{softmax}(V * a_t + c)$

Backpropagation Through Time (BPTT)

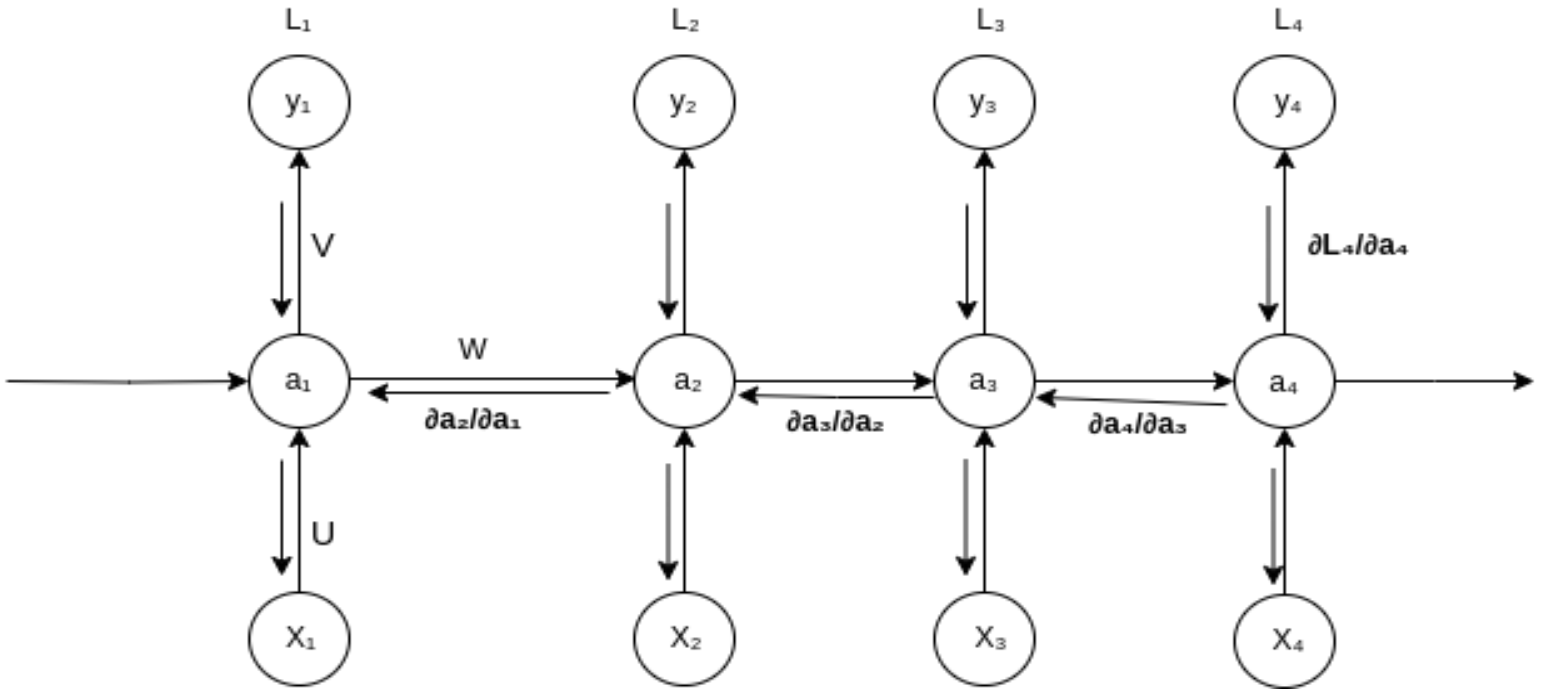
- After processing the entire sequence, RNN generates a sequence of predicted outputs, $\hat{y} = [\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_t]$.
- The relationship between the lengths of the input sequence n and the output sequence t depends on the specific architecture and purpose of the RNN.
- Loss is then computed by comparing predicted output ŷ at each time step with actual target output y. Loss function given by,

$$L(y, \hat{y}) = \frac{1}{t}\sum_{i=1}^{t}(\hat{y}_i - y_i)^2 = \frac{1}{t}\sum_{i=1}^{t}(softmax(V * a_t + c) - y_i)^2$$

Backpropagation Through Time (BPTT)

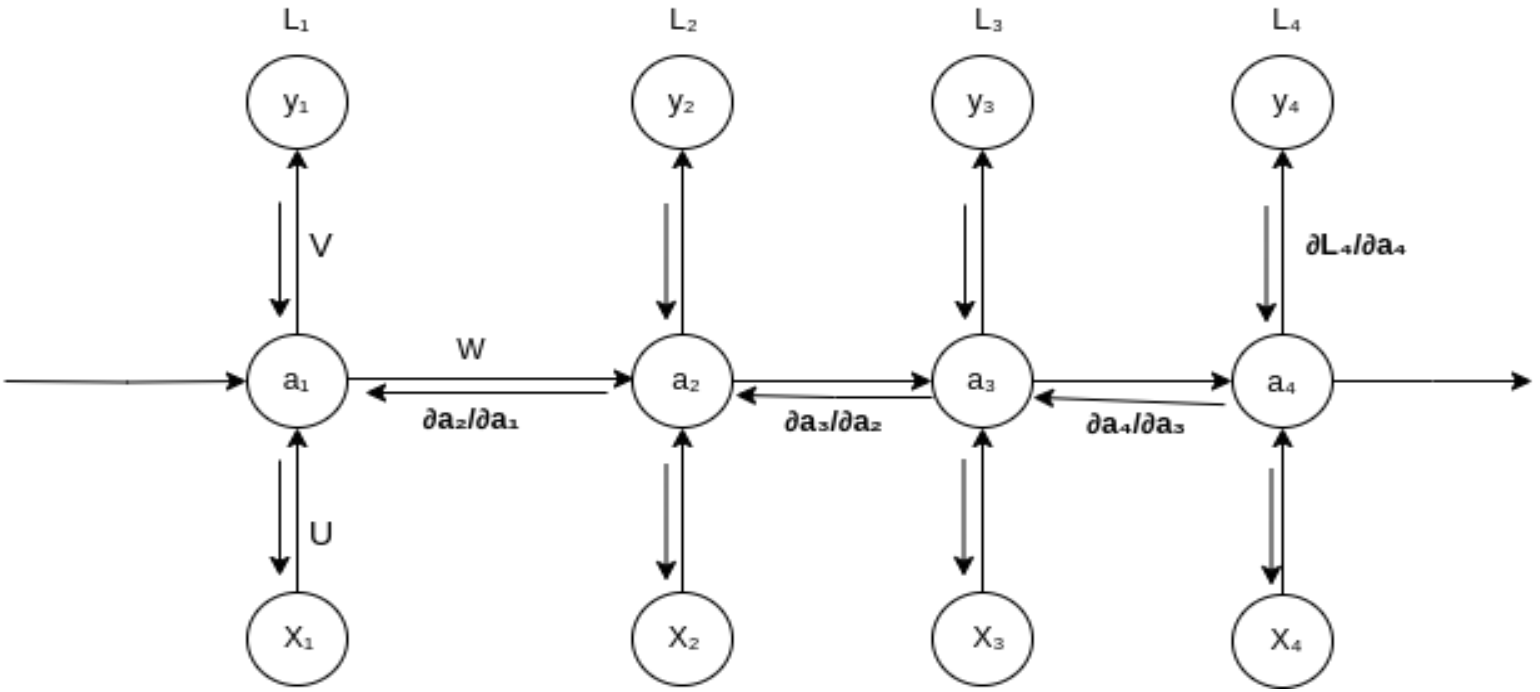
2. **Backward Pass:** The backward pass in BPTT involves computing the gradients of the loss function with respect to the network's parameters (U, W, V and biases) over each time step.

- Let's explore the concept of backpropagation through time by computing the gradients of loss at time step t=4.
- The figure below also serves as an illustration of backpropagation for time step 4.



Backpropagation Through Time (BPTT)

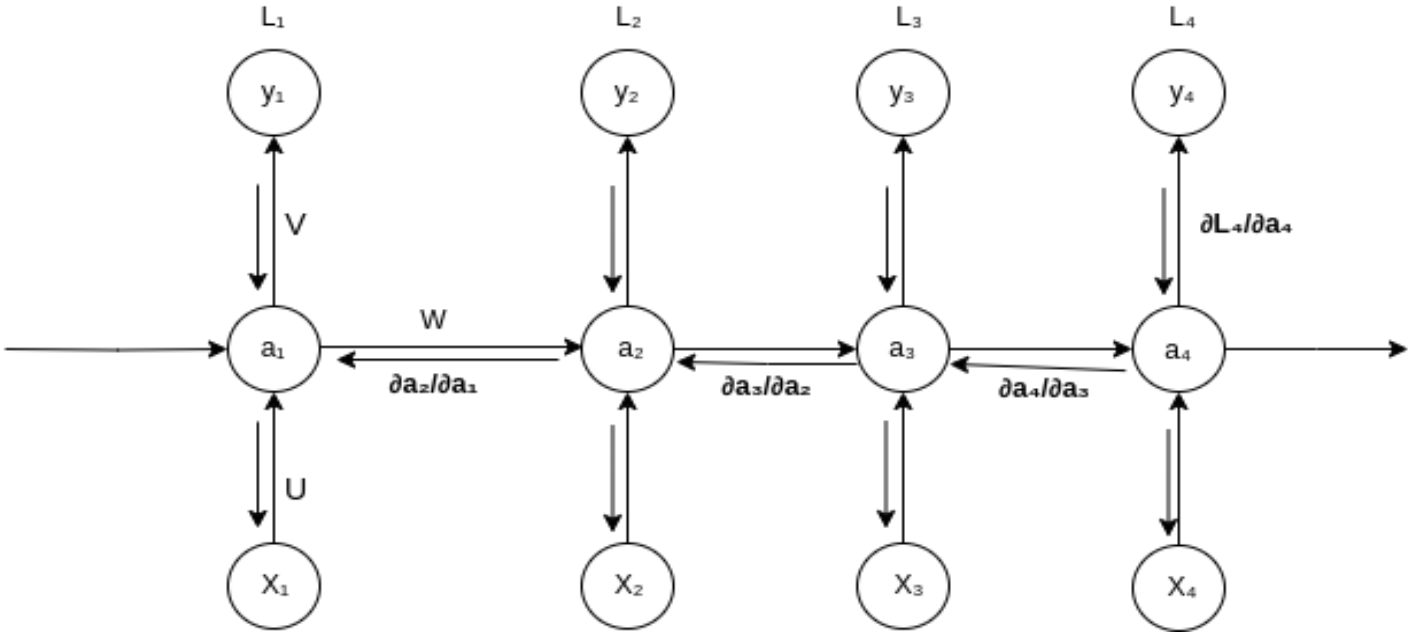
- **Derivative of loss L w.r.t V**
- Loss L is a function of predicted value ŷ, so using the chain rule ∂L/∂V can be written as,
- ∂L/∂V = (∂L/∂ŷ) * (∂ŷ/∂V)



Derivative of loss L w.r.t W

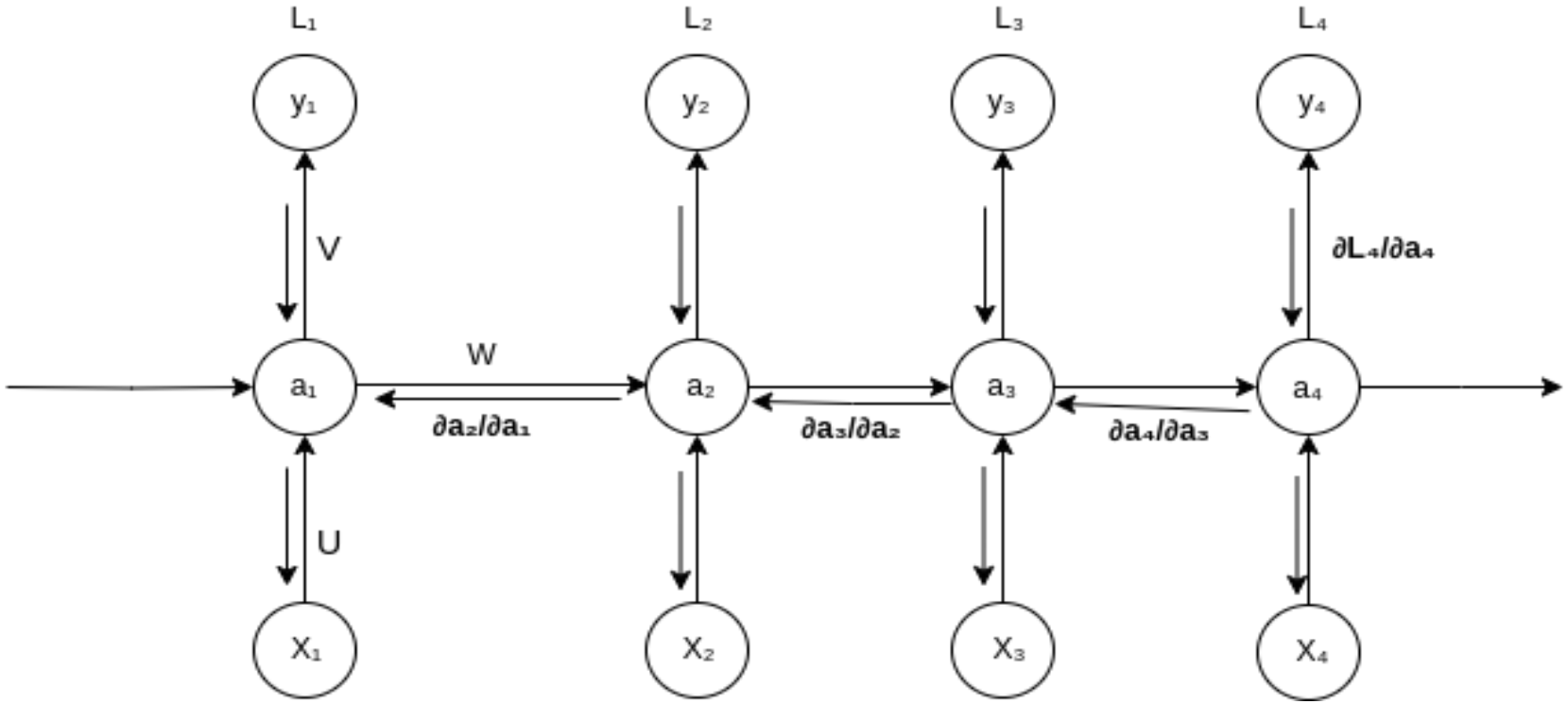
- Applying the chain rule of derivatives ∂L/∂W can be written as follows:

(i) The loss at the 4th time step is dependent upon ŷ due to the fact that the loss is calculated as a function of ŷ, which is in turn dependent on the current time step's hidden state a_4, a_4 is influenced by both W and a_3, and again a_3 is connected to both a_2 and W, and a_2 depends on a_1 and also on W.



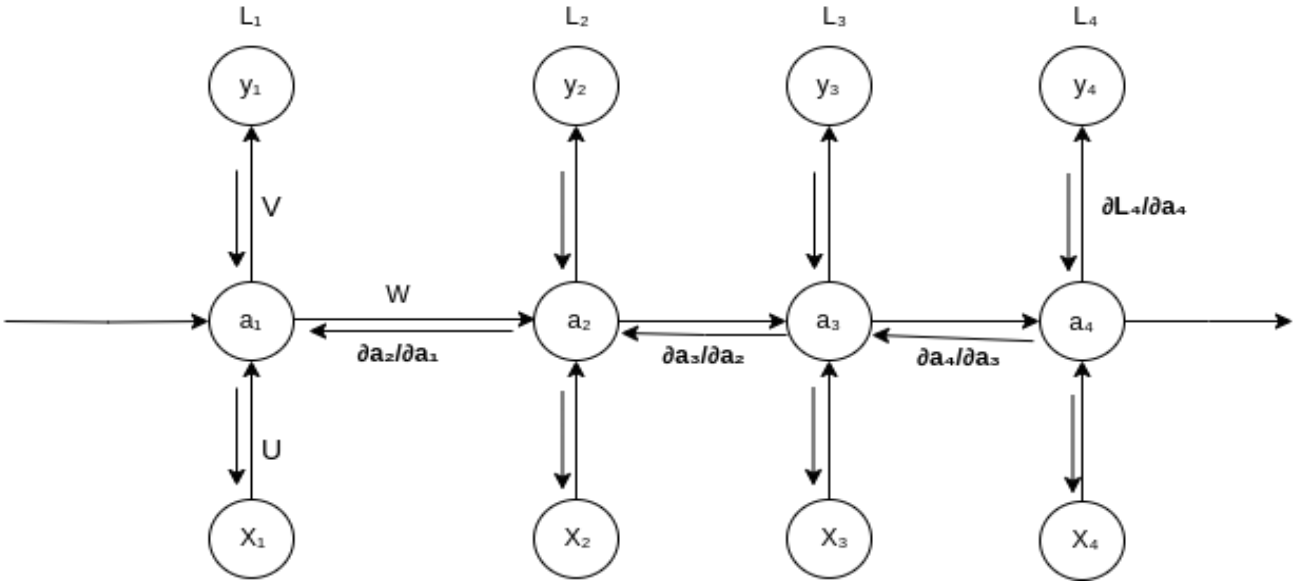
Derivative of loss L w.r.t W

- $\partial L_4/\partial W = (\partial L_4/\partial \hat{y}_4 * \partial \hat{y}_4/\partial a_4 * \partial a_4/\partial W) + (\partial L_4/\partial \hat{y}_4 * \partial \hat{y}_4/\partial a_4 * \partial a_4/\partial a_3 * \partial a_3/\partial W) + (\partial L_4/\partial \hat{y}_4 * \partial \hat{y}_4/\partial a_4 * \partial a_4/\partial a_3 * \partial a_3/\partial a_2 * \partial a_2/\partial W) + (\partial L_4/\partial \hat{y}_4 * \partial \hat{y}_4/\partial a_4 * \partial a_4/\partial a_3 * \partial a_3/\partial a_2 * \partial a_2/\partial a_1 * \partial a_1/\partial W)$



Derivative of loss L w.r.t U

- Similarly, $\partial L/\partial U$ can be written as,
- $\partial L_4/\partial U = (\partial L_4/\partial \hat{y}_4 * \partial \hat{y}_4/\partial a_4 * \partial a_4/\partial U) + (\partial L_4/\partial \hat{y}_4 * \partial \hat{y}_4/\partial a_4 * \partial a_4/\partial a_3 * \partial a_3/\partial U) + (\partial L_4/\partial \hat{y}_4 * \partial \hat{y}_4/\partial a_4 * \partial a_4/\partial a_3 * \partial a_3/\partial a_2 * \partial a_2/\partial U) + (\partial L_4/\partial \hat{y}_4 * \partial \hat{y}_4/\partial a_4 * \partial a_4/\partial a_3 * \partial a_3/\partial a_2 * \partial a_2/\partial a_1 * \partial a_1/\partial U)$



Derivative of loss L w.r.t U

- Where, $L(y, \hat{y}) = \frac{1}{t}\sum_{i=1}^{t}(\hat{y}_i - y_i)^2 = \frac{1}{t}\sum_{i=1}^{t}(softmax(V * a_t + c) - y_i)^2$

  and $a_t = U * X_t + W * a_{t-1} + b$

  $a_t = \sigma(a_t)$ and $\hat{y}_t = softmax(V * a_t + c)$

- Here we're summing up the gradients of loss across all time steps which represents the key difference between BPTT and regular backpropagation approach.

Training through RNN

- A single-time step of the input is provided to the network.
- Then calculate its current state using a set of current input and the previous state.
- The current h_t becomes h_{t-1} for the next time step.
- One can go as many time steps according to the problem and join the information from all the previous states.
- Once all the time steps are completed the final current state is used to calculate the output.
- The output is then compared to the actual output i.e the target output and the error is generated.
- The error is then back-propagated to the network to update the weights and hence the network (RNN) is trained using Backpropagation through time.

Difference between RNN and Simple Neural Network

Recurrent Neural Network	Deep Neural Network
Weights are same across all the layers number of a Recurrent Neural Network	Weights are different for each layer of the network
Recurrent Neural Networks are used when the data is sequential and the number of inputs is not predefined.	A Simple Deep Neural network does not have any special method for sequential data also here the number of inputs is fixed
The Numbers of parameter in the RNN are higher than in simple DNN	The Numbers of Parameter are lower than RNN
Exploding and vanishing gradients is the the major drawback of RNN	These problems also occur in DNN but these are not the major problem with DNN

Applications of RNNs

- **Natural Language Processing**: RNNs are widely used for language modeling, text generation, machine translation, and sentiment analysis.
- **Speech Recognition**: They can model the temporal aspects of speech data, making them effective for transcribing spoken language.
- **Time Series Prediction**: RNNs can analyze and predict future values in time-dependent datasets, such as stock prices or weather forecasting.
- **Music Generation**: RNNs can be trained to compose music by learning from sequences of musical notes.

Limitations of RNN

- During backpropagation, gradients can become too small, leading to the vanishing gradient problem, or too large, resulting in the exploding gradient problem as they propagate backward through time.
- In the case of vanishing gradients, the issue is that the gradient may become too small where the network struggles to capture long-term dependencies effectively. It can still converge during training but it may take a very long time.
- In contrast, in exploding gradient problem, large gradient can lead to numerical instability during training, causing the model to deviate from the optimal solution and making it difficult for the network to converge to global minima.
- To address these problems, variations of RNN like Long-short term memory (LSTM) and Gated Recurrent Unit (GRU) networks have been introduced.

Variants of RNNs

- **Long Short-Term Memory (LSTM)**: LSTMs are a specialized type of RNN designed to combat the vanishing gradient problem, which often hampers the training of standard RNNs. LSTMs introduce mechanisms called gates (input, forget, and output gates) that regulate the flow of information, allowing LSTMs to learn longer dependencies.
- **Gated Recurrent Unit (GRU)**: GRUs are a simpler alternative to LSTMs, combining the input and forget gates into a single update gate. They perform similarly to LSTMs but with fewer parameters, making them computationally more efficient.
- **Bidirectional RNNs**: These networks process the input sequence in both forward and backward directions, allowing the model to capture information from both past and future contexts. This can enhance performance on tasks like text classification and sequence labeling.

Long Short-Term Memory (LSTM) Networks

- LSTMs are a specialized type of Recurrent Neural Network (RNN) designed to address the vanishing gradient problem.
- It is enabled to learn long-range dependencies in sequential data.
- They achieve this through a unique architecture that includes memory cells and gating mechanisms.

Key Components of LSTM

- An LSTM cell contains several components that interact in a structured way to retain relevant information over long sequences, while discarding unnecessary information.

(i) Cell State:

- The core idea of LSTM is the *cell state*, which acts as a conveyor belt running through the entire sequence, enabling the memory to persist over time.
- It can be thought of as the long-term memory of the network, carrying information across many time steps.
- The cell state is modified in a controlled manner by the various gates.

Key Components of LSTM

(ii) Gates:

- LSTMs use three types of gates to control the flow of information:
  - **Forget Gate**: The forget gate determines which information from the previous cell state should be discarded.
  - **Input Gate**: Determines what new information should be added to the cell state.
  - **Output Gate**: Decides what part of the cell state should be output to the next layer.

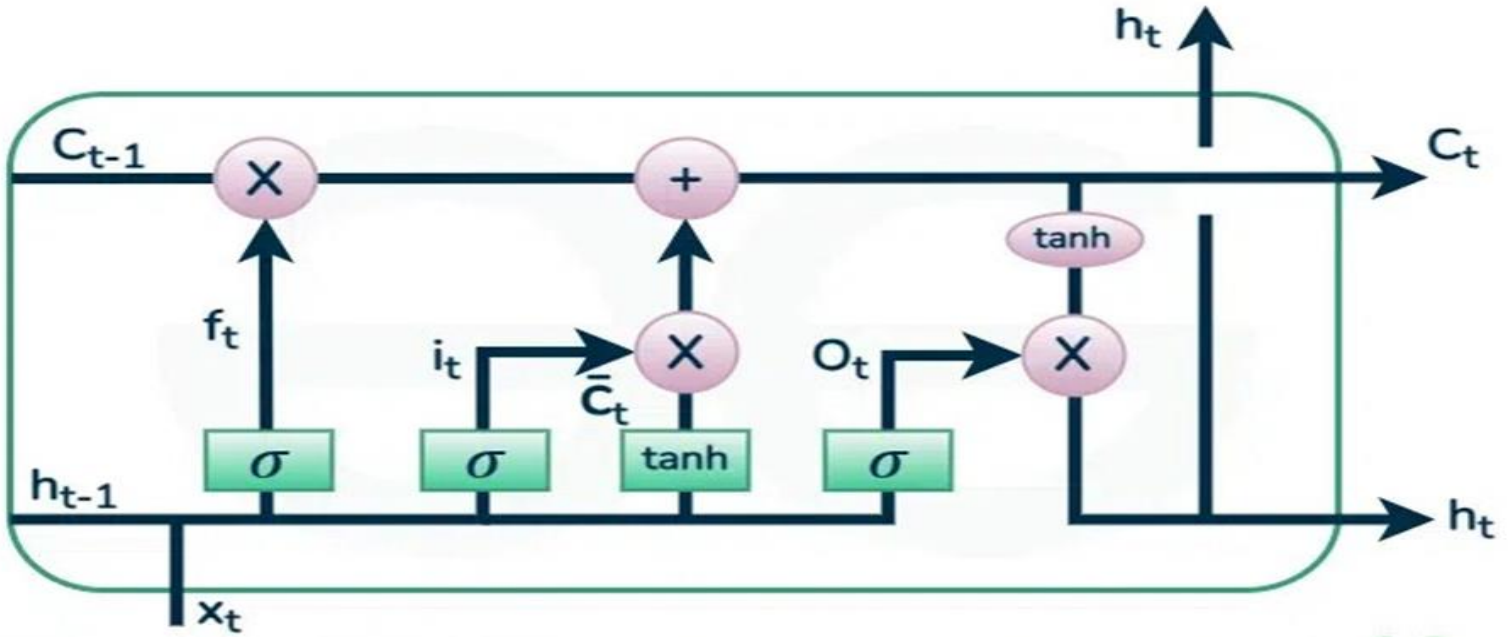
Key Components of LSTM

1. Forget Gate:

- The forget gate determines which information from the previous cell state should be discarded.
- It takes the previous hidden state (h_{t-1}) and the input at the current time step (x_t) as inputs and passes them through a sigmoid activation function.

LSTM Cell Operations

1. **Forget Gate:**
   - Input: Current input x_t and previous hidden state h_{t-1}.
   - Calculation: $f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$
   - Output: A vector of values between 0 and 1 that determine what information to forget from the cell state.
   - The output value 0 means "completely forget" and 1 means "completely retain."



LSTM Cell Operations

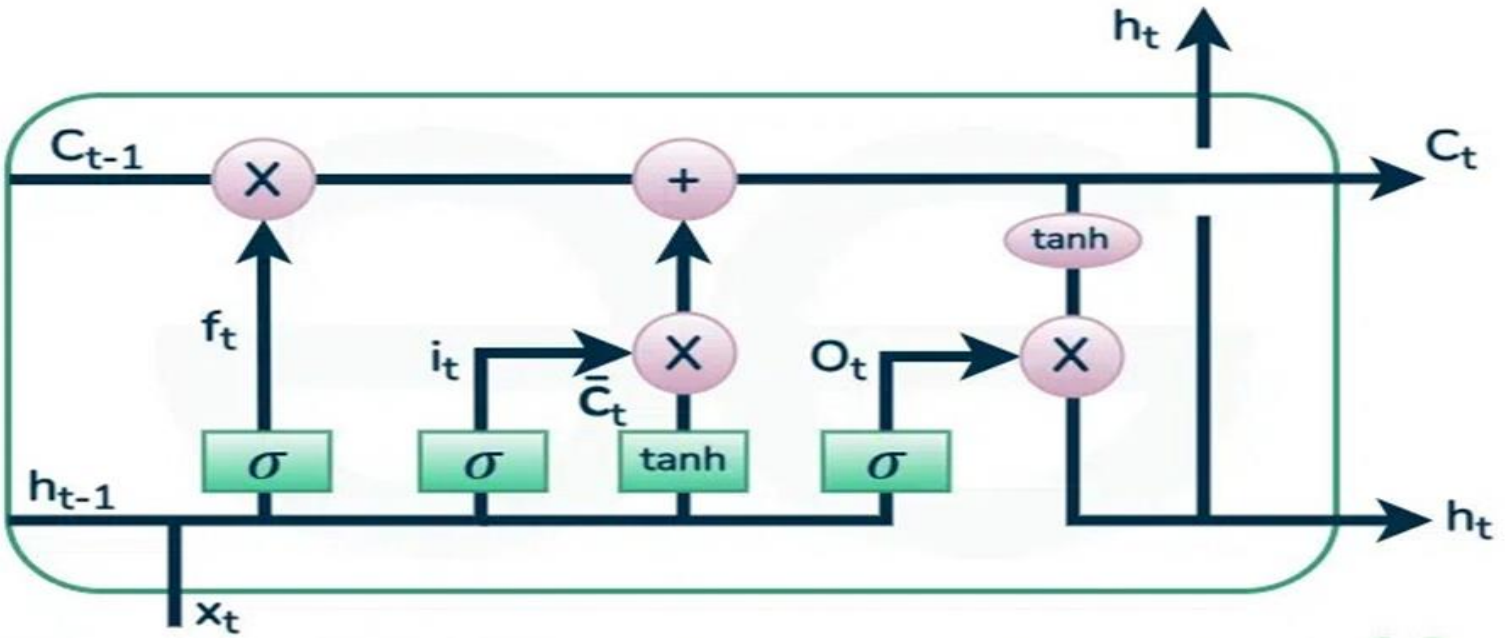
2. Input Gate (i_t):

- The input gate controls how much of the new information from the current input and hidden state should be added to the cell state.
- Like the forget gate, it uses a sigmoid function to output a value between 0 and 1.

2. **Input Gate:**

- Input: Current input x_t and previous hidden state h_{t-1}.
- Calculation:
  - $i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$ (input gate)

- Output: The input gate determines which values to update in the cell state



LSTM Cell Operations

3. Candidate Cell State ($\tilde{C}_t$):

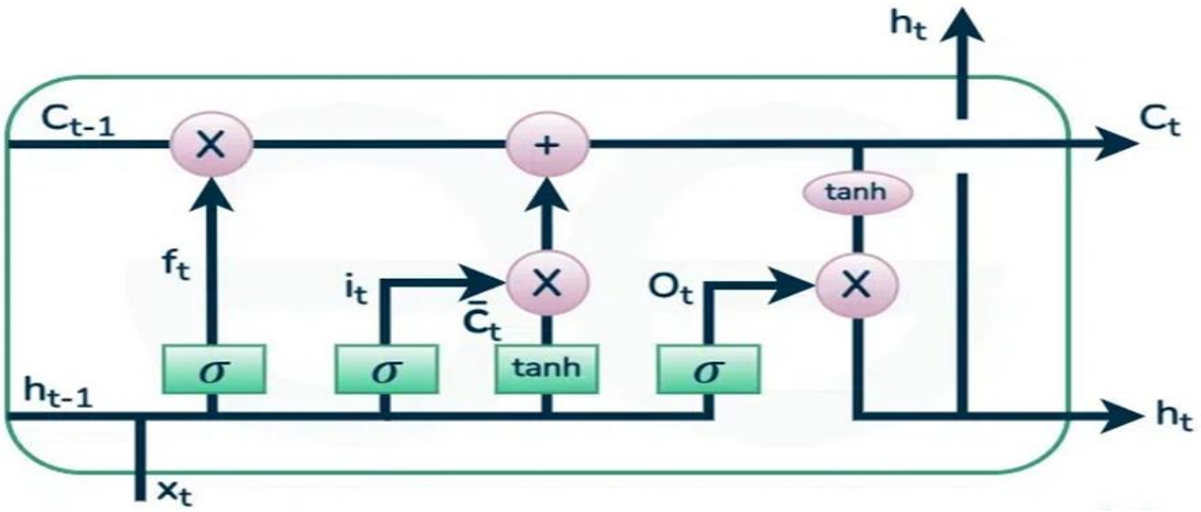
- A tanh layer is used to create a new candidate cell state, which contains potential new information that could be added to the cell state.
- This is generated using the current input and the previous hidden state. The formula is:

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

LSTM Cell Operations

4. **Updating the Cell State (C_t)**:

- The new cell state is updated by combining the forget gate's decision and the input gate's decision.
- The forget gate decides what to remove from the previous cell state, and the input gate decides what new information to add.
- The formula is: $C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$



LSTM Cell Operations

5. Output Gate(O_t):

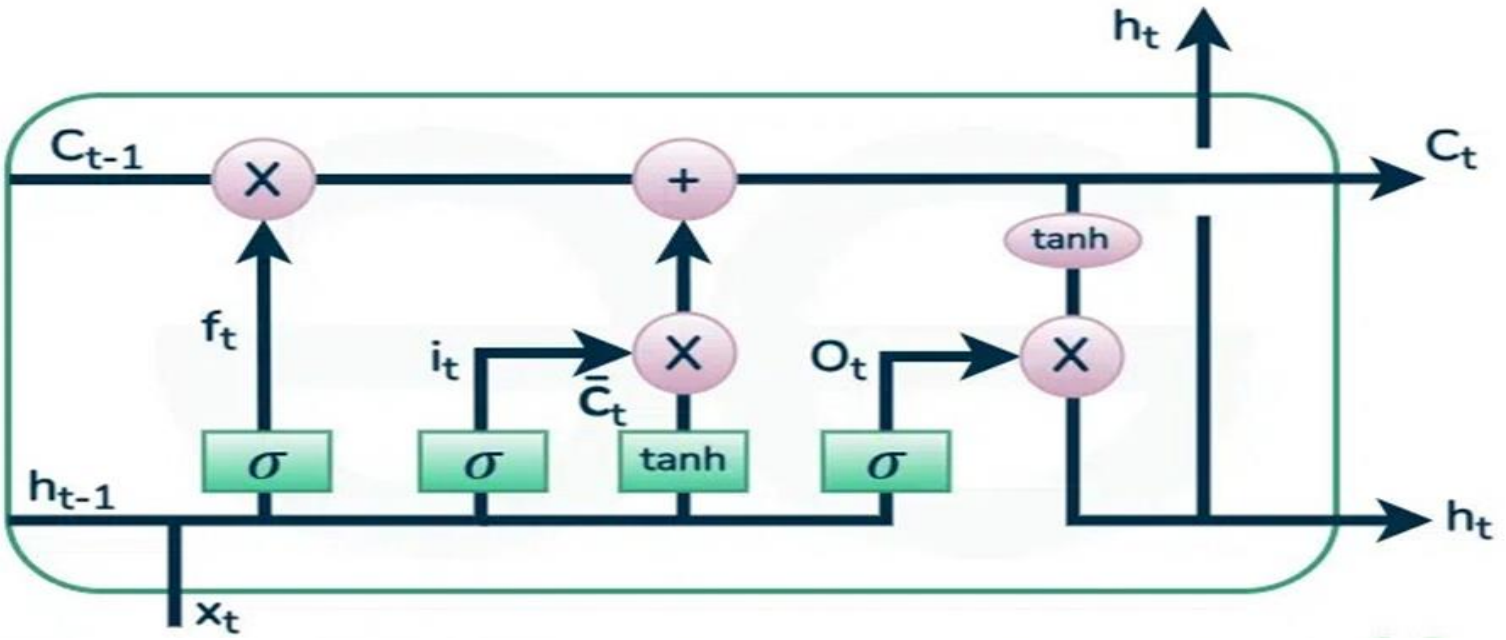
- The output gate decides what the next hidden state (h_t) should be.
- This hidden state is used for predictions and also passed to the next time step.
- The output gate first passes the previous hidden state and the current input through a sigmoid activation to determine which parts of the cell state should be output to the next time step.
- Then, the updated cell state is passed through a tanh function to squash the values between -1 and 1, and the result is multiplied element-wise by the output gate's decision.

Output Gate:

- Input: Current input x_t and new cell state C_t.
- Calculation: $o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$
- Output: The final output is calculated as:

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

- This output will be passed to the next LSTM cell and to the next layer of the network.



LSTM Working Mechanism:

- At each time step t:
- **Forget gate** controls what proportion of the previous cell state C_{t-1} should be "forgotten."
- **Input gate** decides how much of the current input (combined with the previous hidden state) should update the current cell state.
- **Cell state** is updated based on the decisions of the forget and input gates.
- **Output gate** controls the flow of information from the updated cell state to the hidden state h_t, which is passed to the next time step and possibly used as output.

LSTM

- **Key Advantages of LSTM:**

1. **Long-Term Dependency Handling**: LSTMs are designed to maintain information over longer time periods without losing it due to vanishing gradients, unlike standard RNNs.
2. **Selective Memory**: The gating mechanisms allow the network to decide what to remember and what to forget, which makes it suitable for tasks involving sequential data with long-term dependencies (e.g., time series prediction, language modeling, machine translation).

- **LSTM Variants:**

1. **Bidirectional LSTM (BiLSTM)**: Combines two LSTMs, one processing the input sequence forward and another processing it backward. This provides a more comprehensive understanding of context.
2. **Stacked LSTM**: Involves multiple LSTM layers stacked on top of each other, allowing for a more complex and hierarchical feature extraction process.

Input gate Vs Candidate cell state

- The **input gate** and the **candidate cell state** are two distinct components of an LSTM cell, and they serve different roles in managing the flow of information:

1. Input Gate (i_t):

- **Purpose**: The input gate controls how much of the new information (i.e., the candidate cell state) should be added to the cell state. It acts as a filter to decide what proportion of the candidate information will influence the memory.
- **Operation**: It takes the current input x_t and the previous hidden state h_{t-1}, processes them through a **sigmoid function**, and produces an output between 0 and 1. This output is a gate or mask that regulates the flow of information into the cell state.
- Formula:

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

- Where σ is the sigmoid activation function, which outputs values between 0 and 1. A value close to 0 means little or no new information is allowed, while a value close to 1 means new information is allowed in fully.

Input gate Vs Candidate cell state

2. Candidate Cell State ($\tilde{C}_t$):

- **Purpose**: The candidate cell state represents potential new information that could be added to the cell state. It is a processed version of the current input and previous hidden state, designed to offer new content for updating the cell memory.
- **Operation**: It is computed by applying a **tanh activation** function to a weighted combination of the current input x_t and the previous hidden state h_{t-1}. The tanh activation squashes the values between -1 and 1, generating the candidate information that might be useful to update the cell state.
- Formula: $\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$

Different Purpose

- **Input Gate (i_t)**:
- The input gate is a **control mechanism**. It does not directly generate new information.
- Instead, it **decides** how much of the new information (generated by the candidate cell state) should influence the cell state. It acts like a filter or gate.
- It outputs values between 0 and 1 (via a sigmoid activation) to control the flow of the candidate cell state.
- Think of it as a **gate that controls the flow of information**: it's there to allow or restrict how much new information should be considered important enough to update the memory.

Different Purpose

Candidate Cell State ($\tilde{C}_t$):

- The candidate cell state is the actual **new information** that could potentially update the cell's memory. It's not controlling anything—it is **proposing new content** to be added to the memory.
- The candidate cell state represents a set of new values, generated through a tanh function, which compresses the potential values to lie between -1 and 1.
- Think of it as the **potential memory update**: it is suggesting new content that could modify the cell's memory, but it won't have an effect unless the input gate allows it.

Different Activation Functions:

- **Input Gate (i_t)**:
- The input gate uses a **sigmoid activation function** (σ). The sigmoid function squashes the output between 0 and 1. This is important because the gate is designed to control how much of the new information should pass through.
- The sigmoid's output close to 0 means **"don't add this new information"**, while an output close to 1 means **"let this new information fully through."**

Different Activation Functions:

Candidate Cell State ($\tilde{C}_t$):

- The candidate cell state uses a **tanh activation function**. Tanh produces values between -1 and 1, which represents the **actual new content** that could potentially update the memory.
- Tanh compresses the candidate information into a smaller range but allows negative, neutral, and positive values to represent different kinds of updates to the memory.

How They Work Together:

- The **candidate cell state** generates the actual **new information** that could be added to the memory (the values between -1 and 1).
- The **input gate** then uses its output (values between 0 and 1) to **decide how much of this candidate information will actually be added** to the current memory (cell state).
- Together, they interact like this:

$$i_t * \tilde{C}_t$$

  - The candidate cell state ($\tilde{C}_t$) proposes changes to the memory, while the input gate (i_t) decides how much of those changes should be allowed.

How They Work Together:

- Think of the candidate cell state as **raw data** (like a list of new facts or observations), and the input gate as the **editor** deciding how much of that data is relevant or useful enough to be stored. The editor might allow some of the new information in fully, some of it partially, and might discard other parts entirely.

Example: Text Generation

- Let we want to train an LSTM to generate text based on a training dataset of sentences.
- **Data Preparation**: We would prepare the text by tokenizing it into sequences of words, transforming it into numerical representations (e.g., word embeddings).
- **Training**:
  - We feed these sequences into the LSTM one time step at a time.
  - The LSTM learns to predict the next word in the sequence based on previous words, utilizing its cell state and gating mechanisms to remember relevant context.
- **Text Generation**:
  - After training, we can initialize the LSTM with a seed word and generate text by iteratively predicting the next word and feeding it back into the model until we reach the desired length.

Prediction of next word

- Suppose, we want to predict the next word of the sequence of words: “The color of an apple is”.
- we need to understand how the LSTM processes sequential data and maintains context through its memory.
- Step-by-Step LSTM Process is as follows:

1. Input Representation

- Each word in the sentence is converted into a numerical representation, often using techniques like one-hot encoding or word embeddings (e.g., Word2Vec, GloVe). For this example, let's assume we have embedded the words:
  - "The" → $\mathbf{x}_1$
  - "color" → $\mathbf{x}_2$
  - "of" → $\mathbf{x}_3$
  - "an" → $\mathbf{x}_4$
  - "apple" → $\mathbf{x}_5$
  - "is" → $\mathbf{x}_6$

2. **Feeding the Input into the LSTM**:

- The LSTM processes each input word one at a time, maintaining a hidden state and cell state. Let's denote:
  - h_t: hidden state at time step t
  - C_t: cell state at time step t

3. **Processing Each Word**:

- For each word, the LSTM performs the following calculations:
- **Forget Gate**: $f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$
- **Input Gate:**

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

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

- **Cell State Update**: $C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$
- **Output Gate:**

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

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

4. **Sequential Updates**:

- This process is repeated for each word in the sequence:
  - After processing "The" → update h_1 and C_1
  - After processing "color" → update h_2 and C_2
  - After processing "of" → update h_3 and C_3
  - After processing "an" → update h_4 and C_4
  - After processing "apple" → update h_5 and C_5
  - After processing "is" → update h_6 and C_6

Making the Prediction

- Once the LSTM has processed the last input word ("is"), it has a hidden state h_6 that encapsulates the contextual information from the entire sequence.
- To predict the next word ("red"), we typically pass h_6 through a softmax layer:

$$P(y_t|y_{t-1}, y_{t-2}, \ldots, y_1) = \text{softmax}(W_p \cdot h_6 + b_p)$$

- Here, $P(y_t)$ represents the probability distribution over the vocabulary for the predicted next word.

6. **Output Interpretation**:

- The output of the softmax layer gives probabilities for each word in the vocabulary, including "red."
- The word with the highest probability is selected as the predicted next word.

Backpropagation Learning

- In LSTM (Long Short-Term Memory) networks, backpropagation learning works similarly to standard neural networks, but with some additional mechanisms to handle the sequential data and the vanishing/exploding gradient problem. Steps of learning are:

1. Forward Pass

- In the forward pass, the LSTM processes input data through its gates and hidden states. It uses the following components:

(i) Forget Gate: Decides what information to discard from the cell state.

(ii) Input Gate: Determines which new information to store in the cell state.

(iii) Output Gate: Decides what information from the cell state will be output.

(iv) Cell State: This is the "memory" of the LSTM, which is updated at each time step.

- The hidden state h_t at each time step and the cell state c_t are passed to the next time step, along with the output.

Backpropagation Learning

2. Loss Calculation

- After the forward pass, the LSTM calculates the output at each time step, and the error (or loss) is computed.
- The loss typically depends on the task (e.g., Mean Squared Error for regression, or Cross-Entropy Loss for classification).

3. Backpropagation Through Time (BPTT)

- In traditional neural networks, backpropagation calculates gradients of the loss with respect to weights by applying the chain rule of derivatives.
- In LSTM networks, this process is extended to "Backpropagation Through Time" (BPTT) because LSTMs deal with sequences of data (like RNN).

BPTT

- **Gradients are computed for each time step**, and the weights are updated by accumulating the gradients over all time steps in the sequence.
- **Long-term dependencies**: LSTMs use their gating mechanisms (like the forget gate) to retain important information and forget irrelevant details, allowing them to capture long-term dependencies in the data.

4. Gradient Flow and the Vanishing Gradient Problem

- **LSTMs help combat the vanishing gradient problem** (which can cause traditional RNNs to forget long-term dependencies) by using a memory cell that can store information over long time intervals.
- During backpropagation, gradients are passed back through the network, including the cell state.
- The gates in the LSTM ensure that gradients are propagated in a way that mitigates the vanishing gradient problem by controlling the flow of information and gradients.

5. Weight Updates

- Once the gradients are calculated via BPTT, the model's weights are updated using an optimization algorithm like Stochastic Gradient Descent (SGD) or Adam.
- The goal is to minimize the loss function by adjusting the weights in a way that reduces the error.

Summary of the Backpropagation Steps in LSTM

- Compute the forward pass, which involves the cell state and hidden state through the various gates.
- Calculate the loss after the forward pass.
- Use BPTT to propagate the error backward through time, calculating gradients at each step for each gate.
- Update the weights using the gradients.

Vanishing Gradient problem

- Long Short-Term Memory (LSTM) networks are a type of recurrent neural network (RNN) designed to address the vanishing gradient problem, a common issue in traditional RNNs when training long sequences.
- The vanishing gradient problem occurs because, during backpropagation, gradients diminish (or sometimes explode) as they are propagated back through the network layers over many time steps, causing the network to "forget" early parts of sequences.
- LSTM networks tackle this problem with a unique architecture that includes **gates and a cell state**.
- These gates and cell state work together to address the vanishing gradient issue:

Vanishing Gradient problem

1. **Cell State:**

- LSTM introduces a cell state (or memory cell) that allows the network to carry information across many time steps without frequent updates, preserving important information.
- The cell state is a sort of highway for gradients, which minimizes the diminishing effect because it can carry values over time steps relatively unchanged, helping maintain gradients.

Vanishing Gradient problem

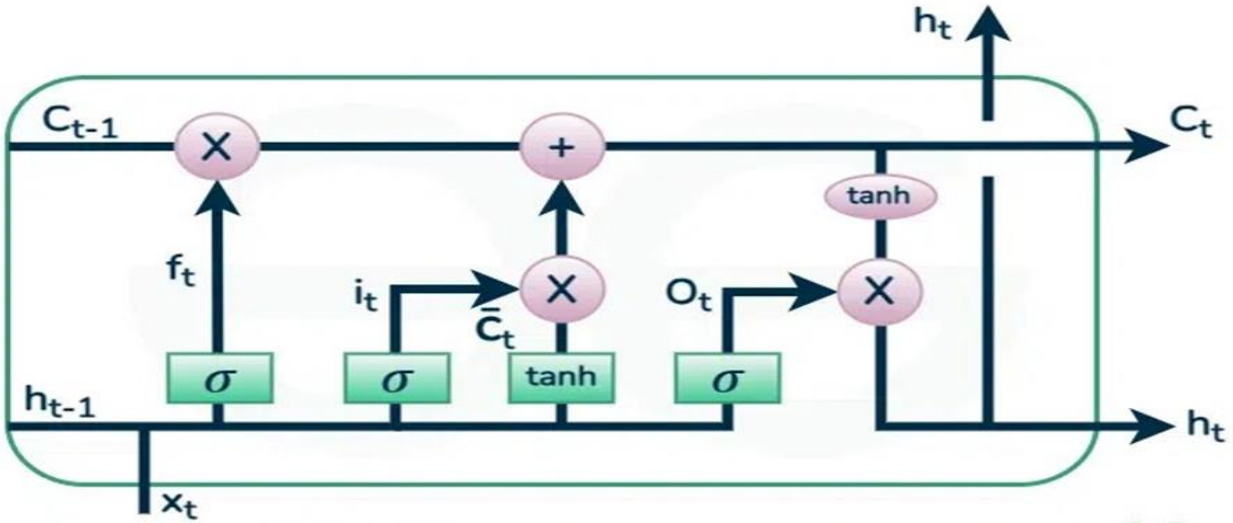
2. **Gates:** LSTMs use three gates — input, forget, and output gates — to regulate information flow. These gates control what information to add, remove, or output at each time step, allowing the network to selectively retain or discard information based on its relevance.

- **Forget Gate:** Decides what part of the cell state should be kept or discarded, allowing the model to “forget” information that is no longer relevant.
- **Input Gate:** Controls how much of the new information should be added to the cell state, helping update it with relevant information without overwhelming it.
- **Output Gate:** Regulates what information from the cell state is used to produce the current output.

Vanishing Gradient problem

3. Gradient Flow Through the Cell State:

- Unlike traditional RNNs, where the gradient directly flows through the hidden state, LSTMs have a more stable pathway through the cell state.
- This setup helps gradients propagate back in time effectively, preserving information over longer sequences and reducing the risk of them vanishing.

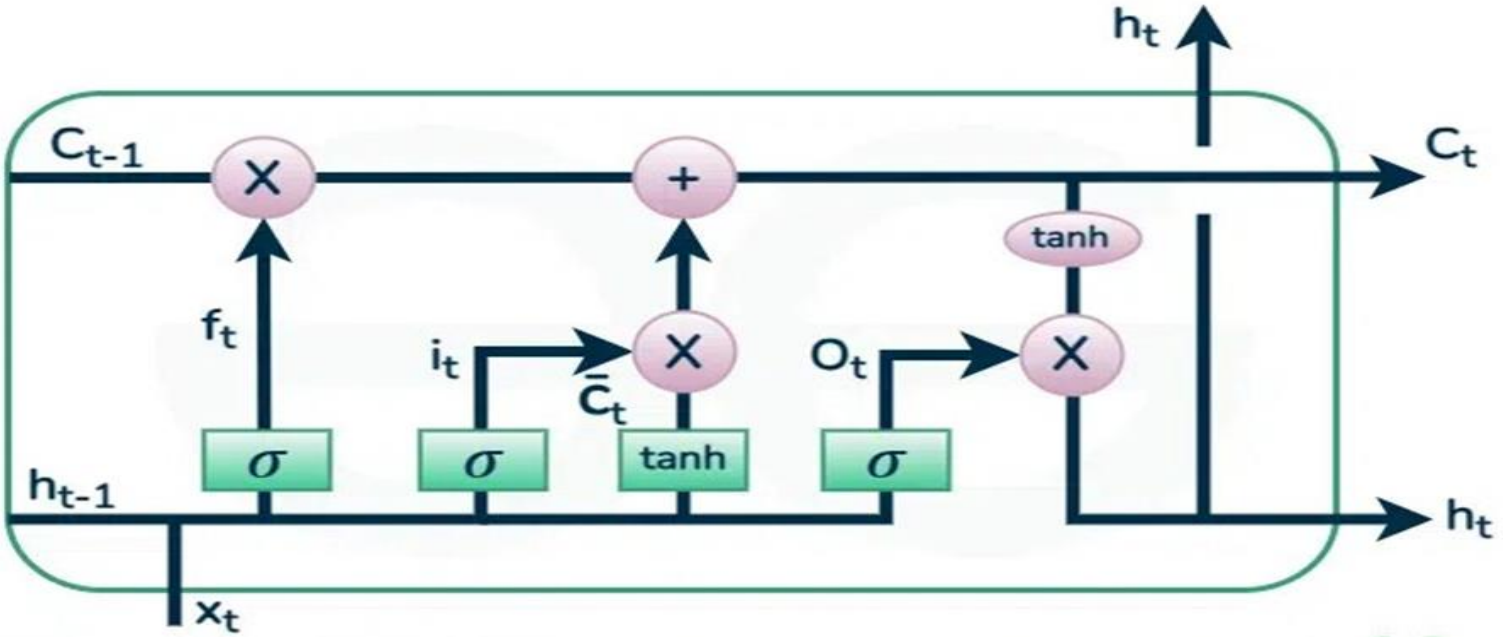


Vanishing Gradient problem

- By managing the flow of information with gates and maintaining a consistent gradient flow through the cell state, LSTMs effectively mitigate the vanishing gradient problem.
- This structure allows them to remember dependencies over longer time sequences than traditional RNNs, making them suitable for tasks where long-term context is essential, like language modeling and time-series analysis.

Vanishing Gradient problem

- LSTMs mitigate the vanishing gradient problem through a careful design of the **cell state** and **gates**, which regulate and stabilize the gradient flow even across many time steps.



- Here's a deeper look at why the LSTM structure prevents the gradient from vanishing:

1. Cell State and Element-Wise Multiplication

- In an LSTM, the cell state flows through time steps almost directly, with only a few controlled, element-wise operations (multiplication by the forget gate and addition from the input gate).
- This design means the cell state can carry values across time steps with minimal modification, maintaining its scale better than in traditional RNNs.
- Because gradients are propagated along this cell state path, they are less likely to vanish over long sequences.
- The forget gate's values are usually set close to 1 (at least for parts of the sequence that need retention), which means the cell state experiences minimal scaling, allowing gradients to stay within a stable range.

2. Forget Gate and Self-Looping Structure

- The forget gate (which multiplies its output with the cell state) is a crucial component. Since the forget gate's output values are typically between 0 and 1, the model learns to retain only relevant information, reducing the risk of unnecessary multiplication that could shrink gradients.
- This self-looping structure is designed so that even after many time steps, the values in the cell state are preserved if the forget gate is close to 1, keeping gradients stable when they propagate back through time.

3. Controlled Update Mechanism

- The LSTM cell state is updated in a controlled way through the input and forget gates, allowing it to accumulate gradients gradually instead of allowing abrupt changes.
- Because of this controlled flow, the gradients are less likely to either explode or vanish. For example, if the cell state is continuously updated with small, relevant amounts of information, it’s unlikely to diminish drastically, maintaining gradient strength across time steps.

4. Activation Functions in Gates

- LSTMs generally use sigmoid and tanh activation functions in gates. Sigmoid activation for gates limits values between 0 and 1, preventing extreme values that might otherwise contribute to exploding or vanishing gradients.
- The tanh activation, used for candidate cell state values, outputs values between -1 and 1, which further stabilizes the update flow into the cell state and reduces the effect of multiplying very small or large numbers over many time steps.

Summary of Why Vanishing Doesn't Happen

- **Gradient Flow Path Stability**: Since gradients flow through a relatively unmodified cell state, they avoid vanishing as they propagate.
- **Controlled Element-Wise Updates**: Gates control each update step, which reduces the chance of gradients shrinking (or exploding) across many time steps.
- **Gate Values Near 1**: The forget gate can be trained to be close to 1 for important information, minimizing multiplicative shrinking of the cell state.
- Together, these mechanisms allow LSTMs to **maintain a more stable gradient flow across long sequences**, which traditional RNNs struggle with due to their simpler, unrestricted architecture.

Thank You!