Introduction to Recurrent Neural Networks

In traditional neural networks, inputs and outputs are treated independently. However, tasks like predicting the next word in a sentence require information from previous words to make accurate predictions. To address this limitation, *Recurrent Neural Networks (RNNs)* were developed.

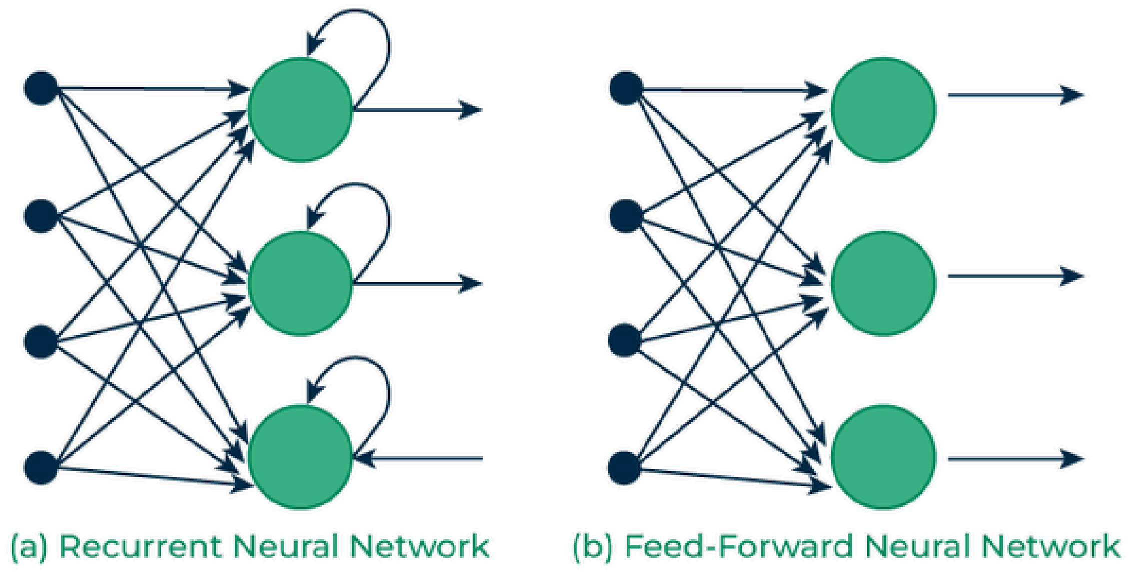
Recurrent Neural Networks introduce a mechanism where the output from one step is fed back as input to the next, allowing them to retain information from previous inputs. This design makes RNNs well-suited for tasks where context from earlier steps is essential, such as predicting the next word in a sentence.

The defining feature of RNNs is their *hidden state*—also called the *memory state*—which preserves essential information from previous inputs in the sequence. By using the same parameters across all steps, RNNs perform consistently across inputs, reducing parameter complexity compared to traditional neural networks. This capability makes RNNs highly effective for sequential tasks.

How RNN Differs from Feedforward Neural Networks

Feedforward Neural Networks (FNNs) process data in one direction, from input to output, without retaining information from previous inputs. This makes them suitable for tasks with independent inputs, like image classification. However, FNNs struggle with sequential data since they lack memory.

Recurrent Neural Networks (RNNs) solve this by incorporating loops that allow information from previous steps to be fed back into the network. This feedback enables RNNs to remember prior inputs, making them ideal for tasks where context is important.

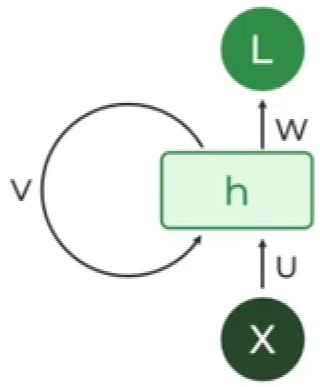


Recurrent Vs Feedforward networks

Key Components of RNNs

1. Recurrent Neurons

The fundamental processing unit in a Recurrent Neural Network (RNN) is a Recurrent Unit, which is not explicitly called a “*Recurrent Neuron.*” Recurrent units hold a hidden state that maintains information about previous inputs in a sequence. Recurrent units can “remember” information from prior steps by feeding back their hidden state, allowing them to capture dependencies across time.

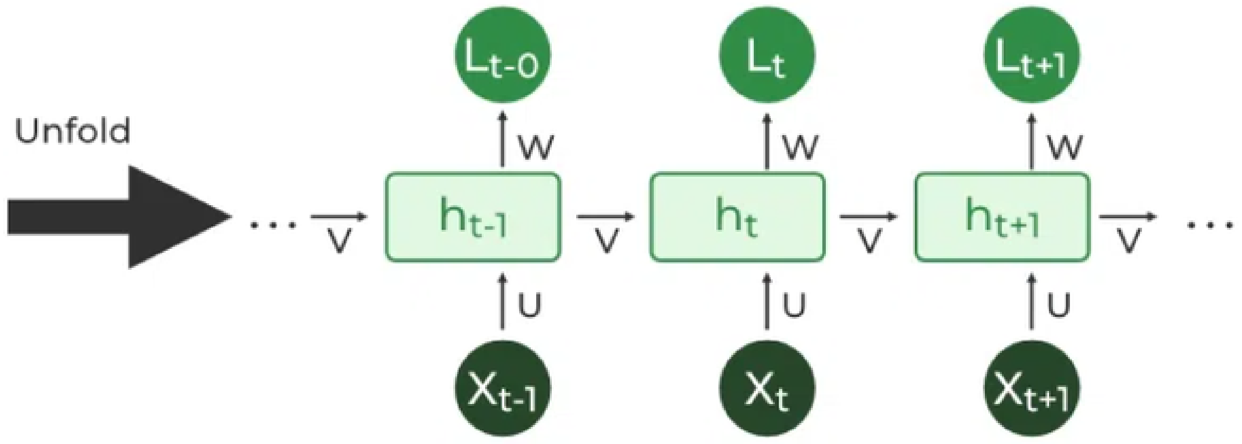


Recurrent Neuron

2. RNN Unfolding

RNN unfolding, or “unrolling,” is the process of expanding the recurrent structure over time steps. During unfolding, each step of the sequence is represented as a separate layer in a series, illustrating how information flows across each time step. This unrolling enables ***backpropagation through time (BPTT)***, a learning process where errors are propagated across time steps to adjust the network’s weights, enhancing the RNN’s ability to learn dependencies within sequential data.

Learn more



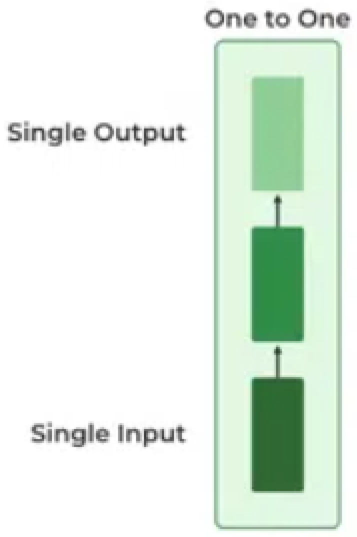
RNN Unfolding

Types Of Recurrent Neural Networks

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

1. One-to-One RNN

One-to-One RNN behaves as the **Vanilla Neural Network**, is the simplest type of neural network architecture. In this setup, there is a single input and a single output. Commonly used for straightforward classification tasks where input data points do not depend on previous elements.

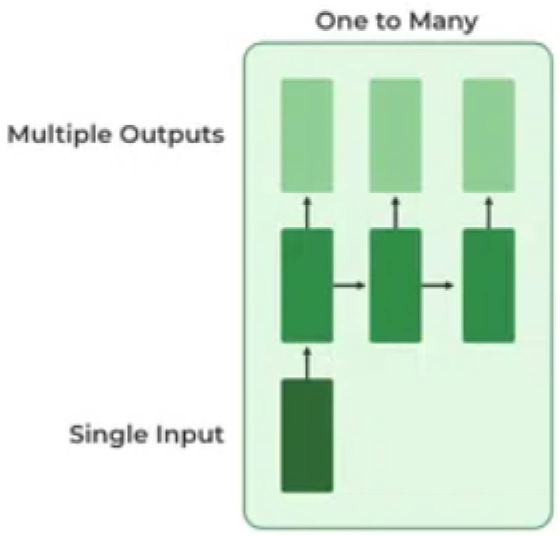


One to One RNN

2. One-to-Many RNN

In a **One-to-Many RNN**, the network processes a single input to produce multiple outputs over time. This setup is beneficial when a single input element should generate a sequence of predictions.

For example, for image captioning task, a single image as input, the model predicts a sequence of words as a caption.

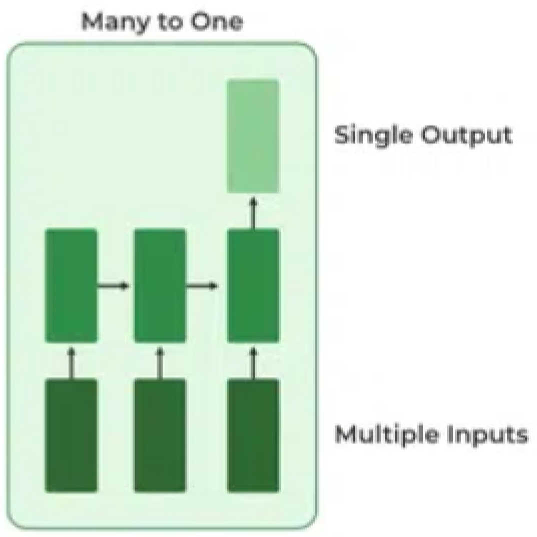


One to Many RNN

3. Many-to-One RNN

The **Many-to-One RNN** receives a sequence of inputs and generates a single output. This type is useful when the overall context of the input sequence is needed to make one prediction.

In sentiment analysis, the model receives a sequence of words (like a sentence) and produces a single output, which is the sentiment of the sentence (positive, negative, or neutral).

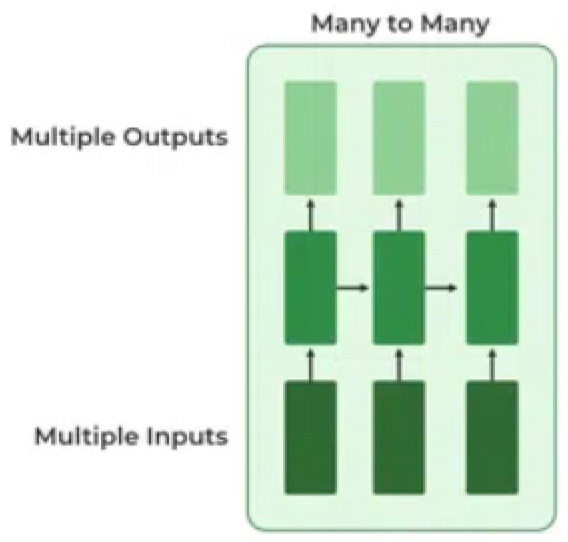


Many to One RNN

4. Many-to-Many RNN

The **Many-to-Many RNN** type processes a sequence of inputs and generates a sequence of outputs. This configuration is ideal for tasks where the input and output sequences need to align over time, often in a one-to-one or many-to-many mapping.

In language translation task, a sequence of words in one language is given as input, and a corresponding sequence in another language is generated as output.



Many to Many RNN

Code: integer_encoding_technique.ipynb

LSTM

What is LSTM – Long Short Term Memory?

LSTM excels in sequence prediction tasks, capturing long-term dependencies. Ideal for time series, machine translation, and speech recognition due to order dependence.

What is LSTM?

Long Short-Term Memory is an improved version of recurrent neural network.

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 model** address this problem by introducing a memory cell, which is a container that can hold information for an extended period.

LSTM architectures 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.

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.

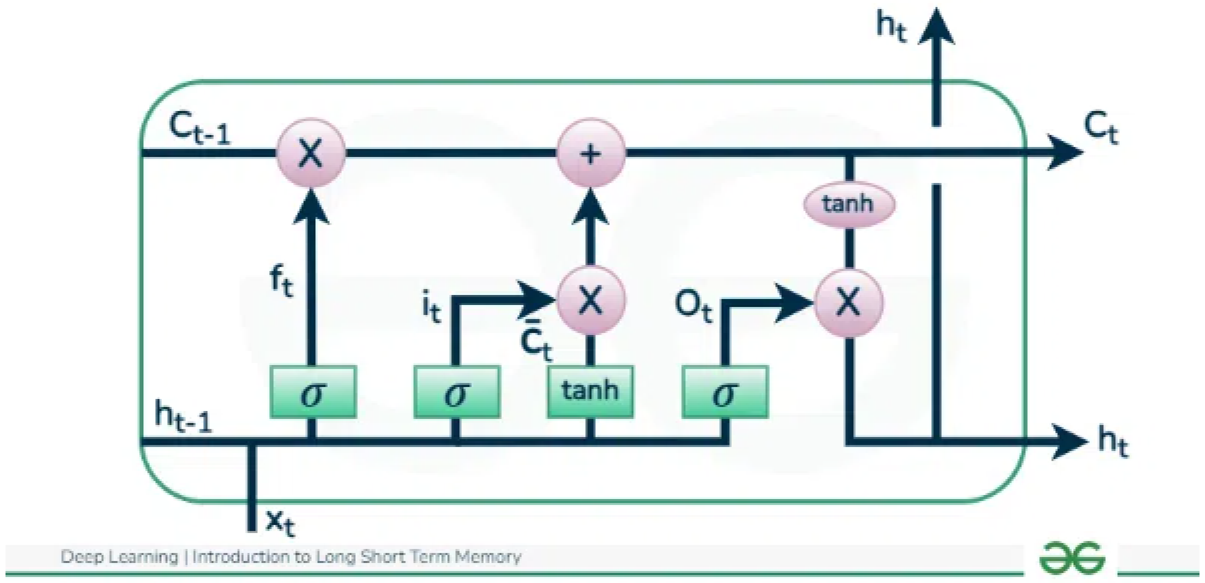
- The output gate controls what information is output from the memory cell.

This allows LSTM networks to selectively retain or discard information as it flows through the network, which allows them to learn long-term dependencies.

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.

LSTM Working

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 –

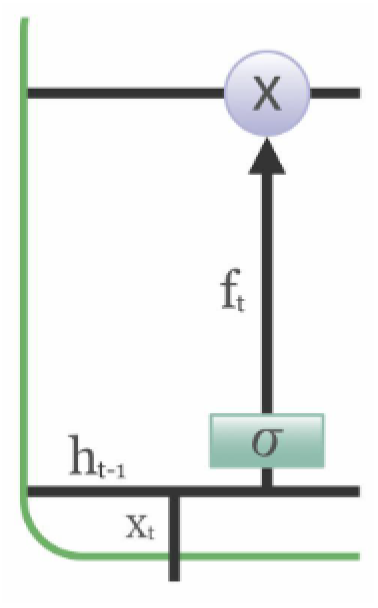
Forget Gate

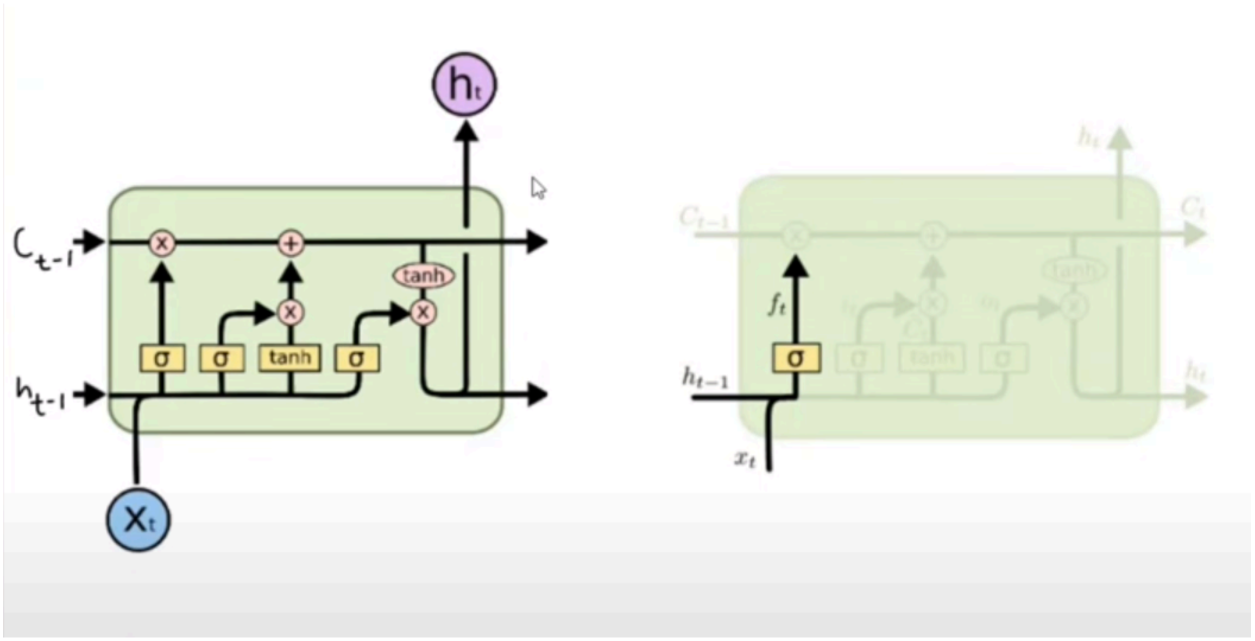
The information that is no longer useful in the cell state is removed with the forget gate. Two inputs x_t (input at the particular time) and h_{t-1} (previous cell output) are fed to the gate and multiplied with weight matrices followed by the addition of bias. The resultant is passed through an activation 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. The equation for the forget gate is:

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

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:

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

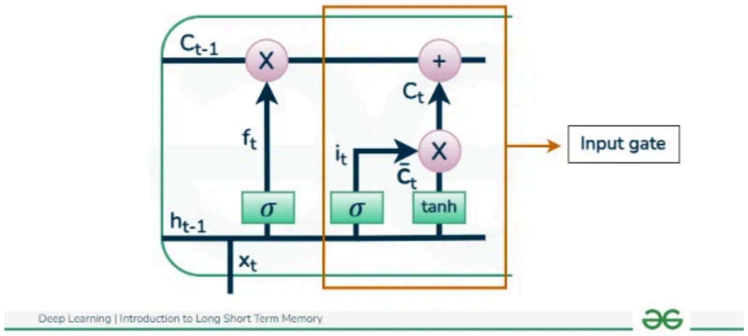
$\hat{C}_t = tanh(W_c \cdot [h_{t-1}, x_t] + b_c)$

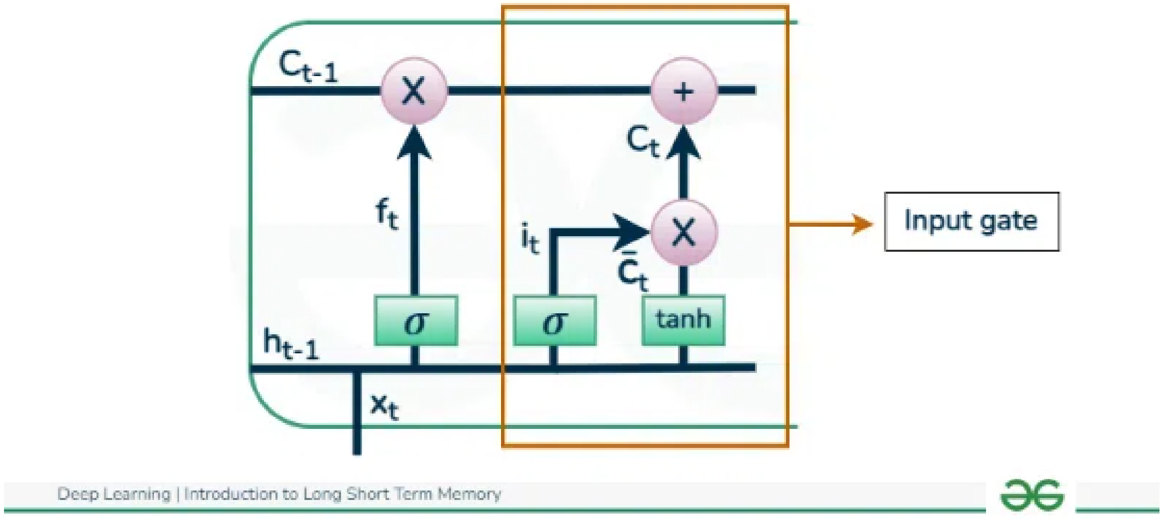
We multiply the previous state by f_t, disregarding the information we had previously chosen to ignore. Next, we include $i_t * C_t$. This represents the updated candidate values, adjusted for the amount that we chose to update each state value.

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

where

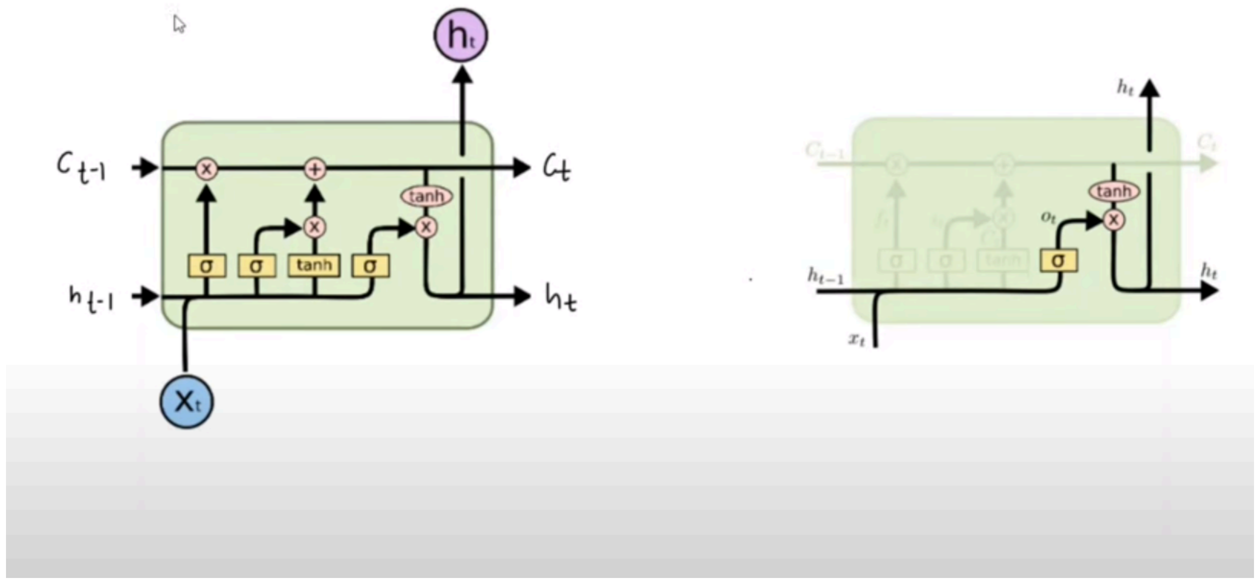
- $\odot$ denotes element-wise multiplication
- tanh is tanh activation function

- 



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:



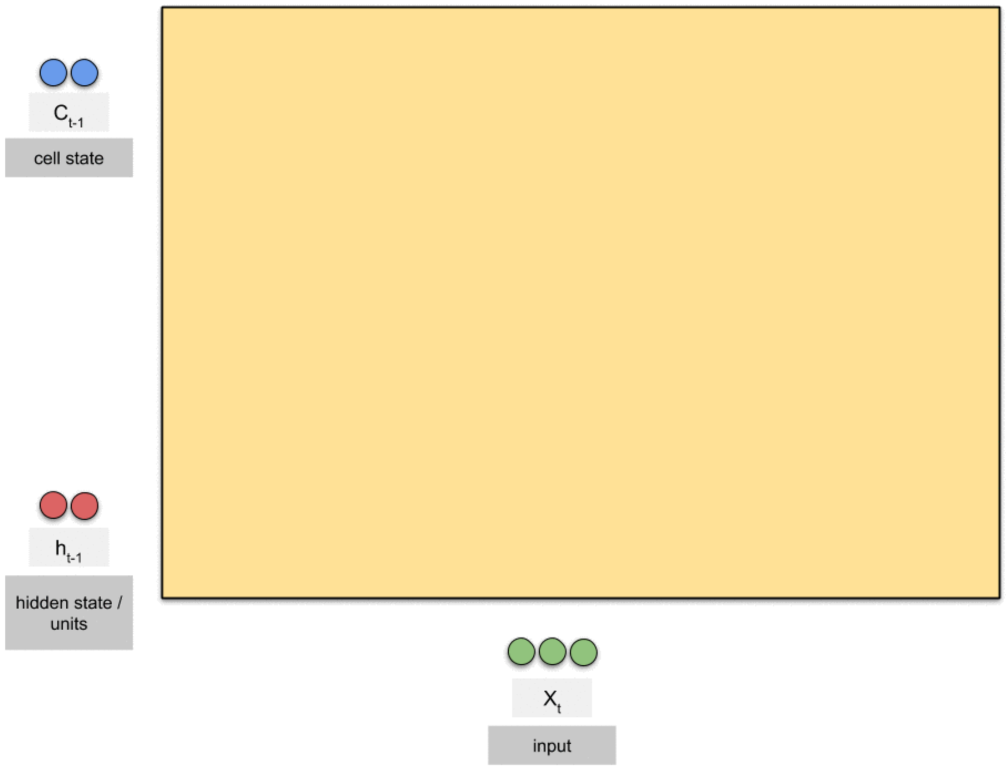
Applications of LSTM

Some of the famous applications of LSTM includes:

- **Language Modeling:** LSTMs have been used for natural language processing tasks such as language modeling, machine translation, and text summarization. They can be trained to generate coherent and grammatically correct sentences by learning the dependencies between words in a sentence.

- **Speech Recognition:** LSTMs have been used for speech recognition tasks such as transcribing speech to text and recognizing spoken commands. They can be trained to recognize patterns in speech and match them to the corresponding text.
- **Time Series Forecasting:** LSTMs have been used for time series forecasting tasks such as predicting stock prices, weather, and energy consumption. They can learn patterns in time series data and use them to make predictions about future events.
- **Recommender Systems:** LSTMs have been used for recommendation tasks such as recommending movies, music, and books. They can learn patterns in user behavior and use them to make personalized recommendations.

Demo: LSTM_DEMO.mp4


C_{t-1}
cell state
h_{t-1}
hidden state / units
X_t
input