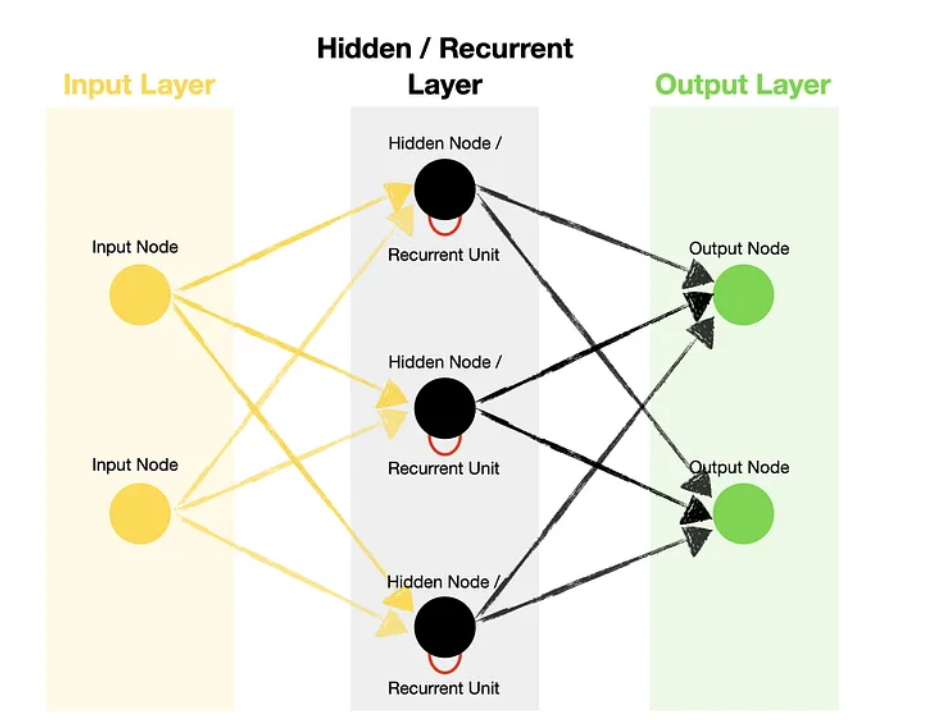
**LSTM Recurrent Neural Networks.**

Standard Recurrent Neural Networks (RNNs) suffer from short-term memory due to a vanishing gradient problem that emerges when working with longer data sequences.

Luckily, we have more advanced versions of RNNs that can preserve important information from earlier parts of the sequence and carry it forward. The two best-known versions are **Long Short-Term Memory (LSTM)** and [**Gated Recurrent Units (GRU)**](https://towardsdatascience.com/gru-recurrent-neural-networks-a-smart-way-to-predict-sequences-in-python-80864e4fe9f6).

# What makes LSTM different from standard RNNs and how does LSTM work?

Let’s start with a quick recap of a simple RNN structure. RNN consists of multiple layers similar to a Feed-Forward Neural Network: the input layer, hidden layer(s), and output layer.



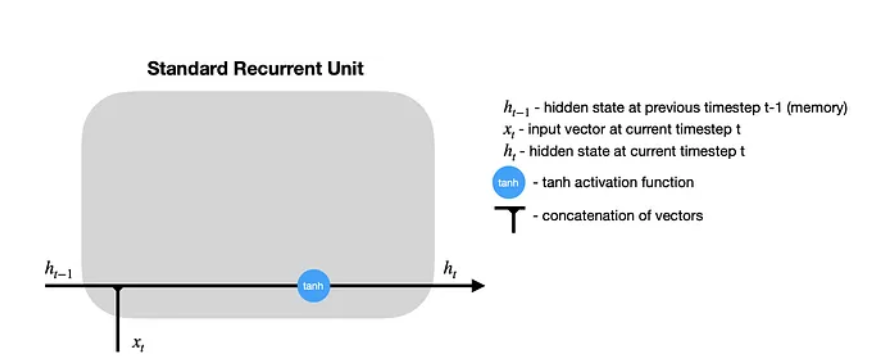
However, RNN contains recurrent units in its hidden layer, which allows the algorithm to process sequence data. It does it by recurrently passing a hidden state from a previous timestep and combining it with an input of the current one.

Timestep — single processing of the inputs through the recurrent unit. The number of timesteps is equal to the length of the sequence.

## **How does LSTM differ from standard RNN?**

We know that RNNs utilize **recurrent units** to learn from the sequence data. So do LSTMs. However, what happens inside the recurrent unit is very different between the two.

Looking inside the simplified recurrent unit diagram of a standard RNN (weights and biases not shown), we notice that there are only two major operations: combining the previous hidden state with the new input and passing it through the activation function:

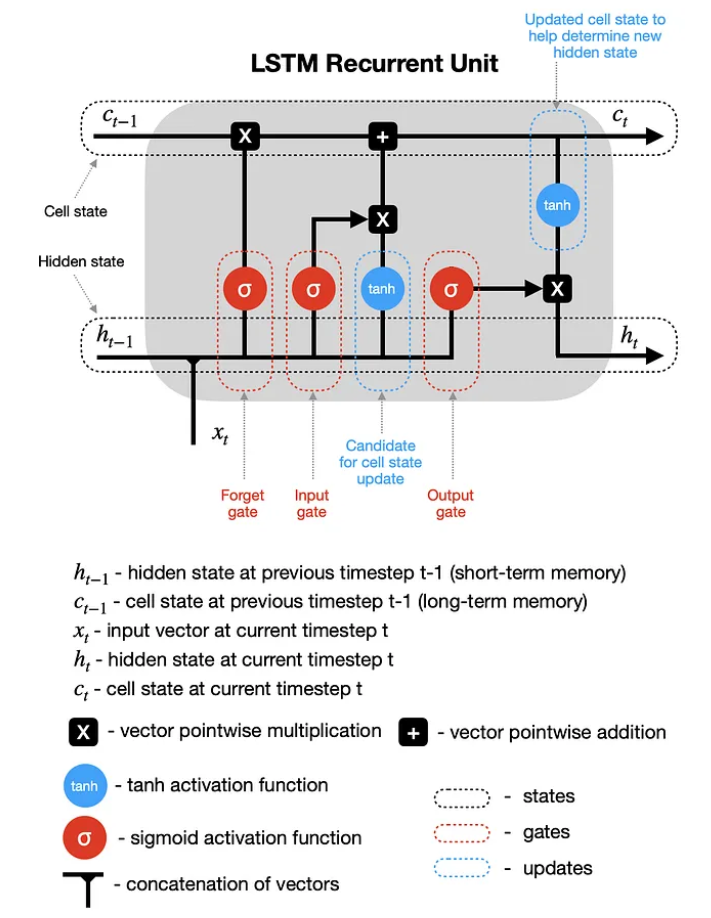


After the hidden state is calculated at timestep t, it is **passed back to the recurrent unit** and combined with the input at timestep t+1 to calculate the new hidden state at timestep t+1. This process repeats for t+2, t+3, …, t+n until the predefined number (n) of timesteps is reached.

Meanwhile, LSTM employs various gates to decide what information to keep or discard. Also, it adds a **cell state**, which is like a long-term memory of LSTM. So, let’s take a closer look:

## **How does LSTM work?**

LSTM recurrent unit is much more complex than that of RNN, which improves learning but requires more computational resources:



**It is important to note that Cell states are part of the long-term memory of the algorithm.**

Let’s go through the simplified diagram (weights and biases not shown) to learn how LSTM recurrent unit processes information.

1. **Hidden state & new inputs**— hidden state from a previous timestep (h\_t-1) and the input at a current timestep (x\_t) are combined before passing copies of it through various gates.
2. **Forget gate**— this gate controls what information should be forgotten. Since the sigmoid function ranges between 0 and 1, it sets which values in the cell state should be discarded (multiplied by 0), remembered (multiplied by 1), or partially remembered (multiplied by some value between 0 and 1).
3. **Input gate**helps to identify important elements that need to be added to the cell state. Note that the results of the input gate get multiplied by the cell state candidate, with only the information deemed important by the input gate being added to the cell state.
4. **Update cell state** —first, the previous cell state (c\_t-1) gets multiplied by the results of the forget gate. Then we add new information from [input gate × cell state candidate] to get the latest cell state (c\_t).
5. **Update hidden state** — the last part is to update the hidden state. The latest cell state (c\_t) is passed through the tanh activation function and multiplied by the results of the output gate.

Finally, the latest cell state (c\_t) and the hidden state (h\_t) go back into the recurrent unit, and the **process repeats at timestep t+1**. The loop continues until we reach the end of the sequence.

We could use LSTMs in four different ways:

* **One-to-one** — theoretically possible, but given one item is not a sequence, you don’t get any benefits offered by LSTMs. Hence, it is better to use a [Feed-Forward Neural Network](https://towardsdatascience.com/feed-forward-neural-networks-how-to-successfully-build-them-in-python-74503409d99a) in such a scenario instead.
* **Many-to-one** — using a sequence of values to predict the next value
* **One-to-many** — using one value to predict a sequence of values.
* **Many-to-many**— using a sequence of values to predict the next sequence of values.