RNN

What Is a Neural Network?

A Neural Network consists of different layers connected to each other, working on the structure and function of a human brain. It learns from huge volumes of data and uses complex algorithms to train a neural net.

Here is an example of how neural networks can identify a dog's breed based on their features.

- The image pixels of two different breeds of dogs are fed to the input layer of the neural network.
- The image pixels are then processed in the hidden layers for feature extraction.
- The output layer produces the result to identify if it's a German Shepherd or a Labrador.
- Such networks do not require memorizing the past output.

Several neural networks can help solve different business problems. Let's look at a few of them.

- Feed-Forward Neural Network: Used for general Regression and Classification problems.
- Convolutional Neural Network: Used for object detection and image classification.
- Deep Belief Network: Used in healthcare sectors for cancer detection.
- RNN: Used for speech recognition, voice recognition, time series prediction, and natural language processing.

What Is a Recurrent Neural Network (RNN)?

RNN works on the principle of saving the output of a particular layer and feeding this back to the input in order to predict the output of the layer.

Below is how you can convert a Feed-Forward Neural Network into a Recurrent Neural Network:

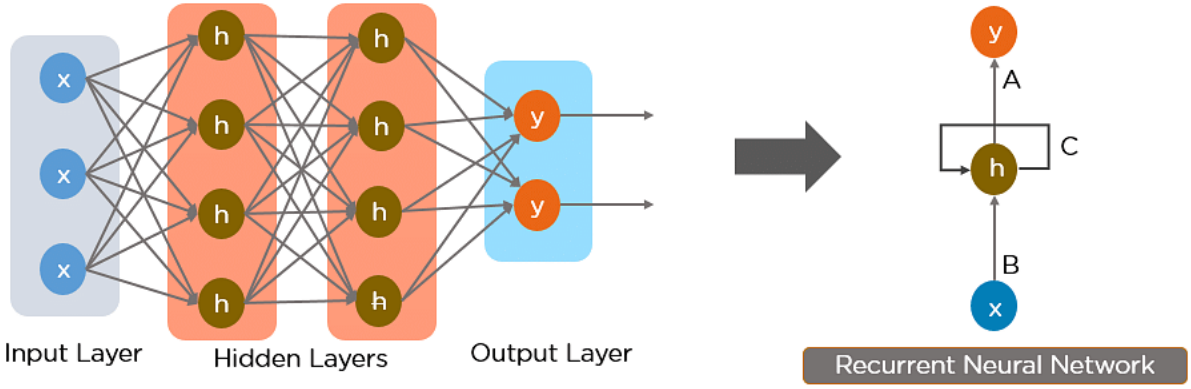


Fig: Simple Recurrent Neural Network

The nodes in different layers of the neural network are compressed to form a single layer of recurrent neural networks. A, B, and C are the parameters of the network.

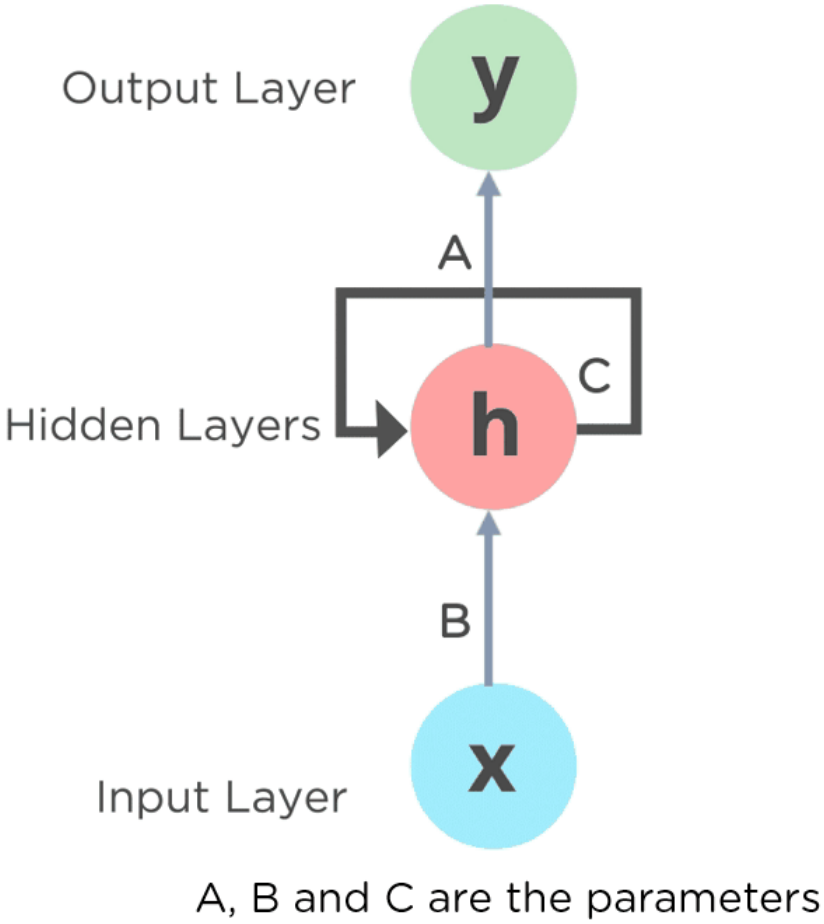


Fig: Fully connected Recurrent Neural Network

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. 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.

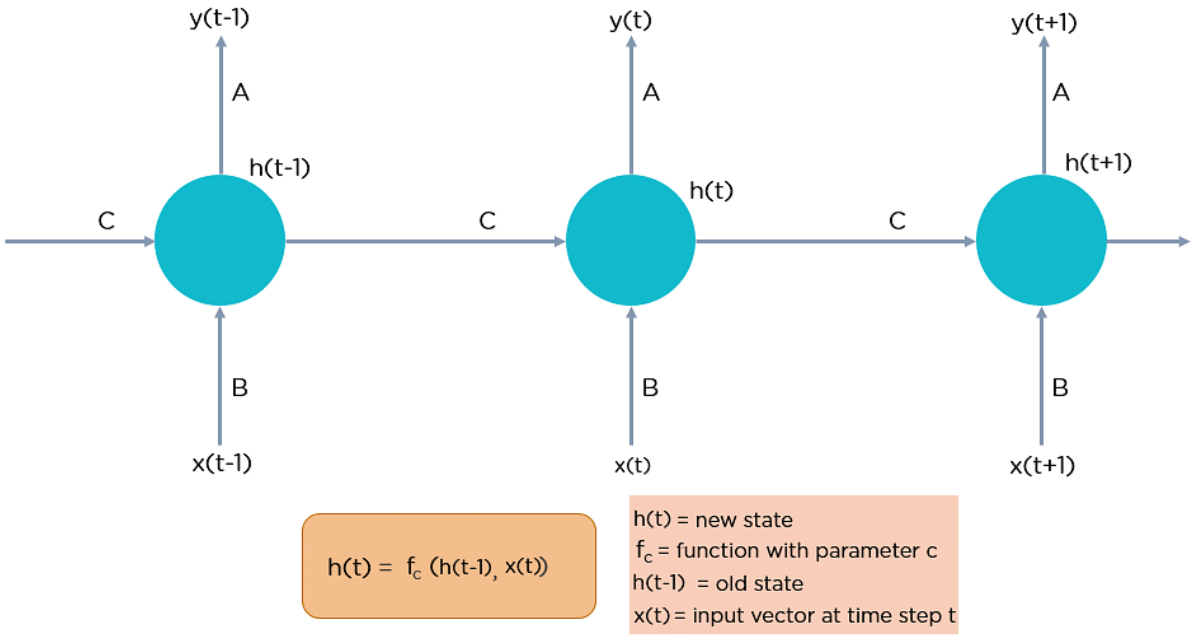


Fig: Fully connected Recurrent Neural Network

Now that you understand what a recurrent neural network is let's look at the different types of recurrent neural networks.

Why Recurrent Neural Networks?

RNN were created because there were a few issues in the feed-forward neural network:

- Cannot handle sequential data
- Considers only the current input
- Cannot memorize previous inputs

The solution to these issues is the RNN. An RNN can handle sequential data, accepting the current input data, and previously received inputs. RNNs can memorize previous inputs due to their internal memory.

How Does Recurrent Neural Networks Work?

In Recurrent Neural networks, the information cycles through a loop to the middle hidden layer.

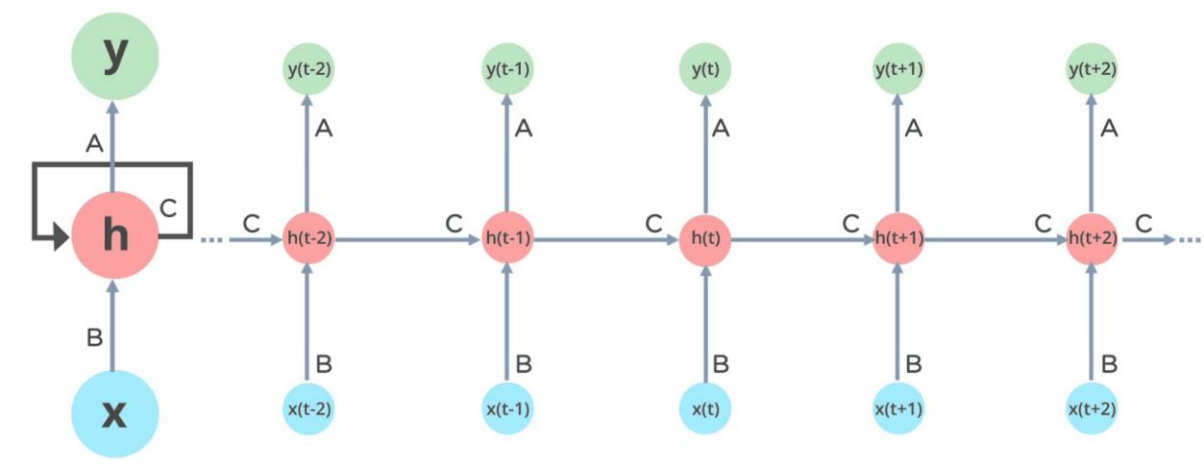


Fig: Working of Recurrent Neural Network

The input layer ‘x’ takes in the input to the neural network and processes it and passes it onto the middle layer.

The middle layer ‘h’ can consist of multiple hidden layers, each with its own activation functions and weights and biases. If you have a neural network where the various parameters of different hidden layers are not affected by the previous layer, ie: the neural network does not have memory, then you can use a recurrent neural network.

The Recurrent Neural Network will standardize the different activation functions and weights and biases so that each hidden layer has the same parameters. Then, instead of creating multiple hidden layers, it will create one and loop over it as many times as required.

Feed-Forward Neural Networks vs Recurrent Neural Networks

A feed-forward neural network allows information to flow only in the forward direction, from the input nodes, through the hidden layers, and to the output nodes. There are no cycles or loops in the network.

Below is how a simplified presentation of a feed-forward neural network looks like:

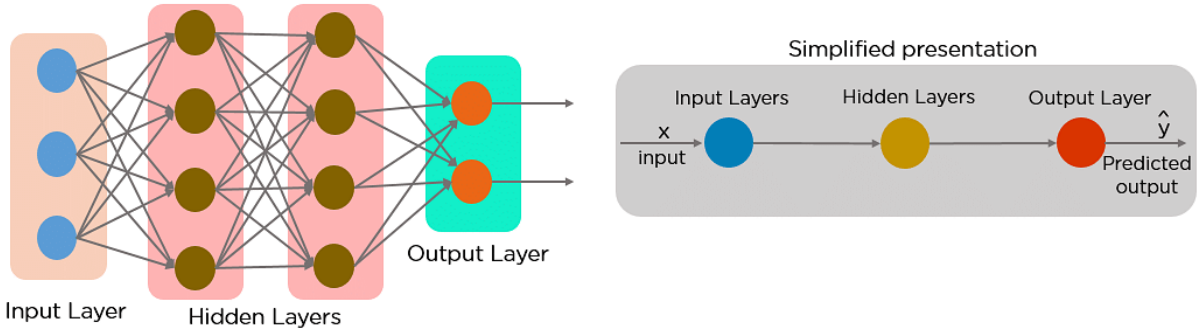


Fig: Feed-forward Neural Network

In a feed-forward neural network, the decisions are based on the current input. It doesn't memorize the past data, and there's no future scope. Feed-forward neural networks are used in general regression and classification problems.

Applications of Recurrent Neural Networks

Image Captioning

RNNs are used to caption an image by analyzing the activities present.

Time Series Prediction

Any time series problem, like predicting the prices of stocks in a particular month, can be solved using an RNN.

Natural Language Processing

Text mining and Sentiment analysis can be carried out using an RNN for Natural Language Processing (NLP).

Machine Translation

Given an input in one language, RNNs can be used to translate the input into different languages as output.

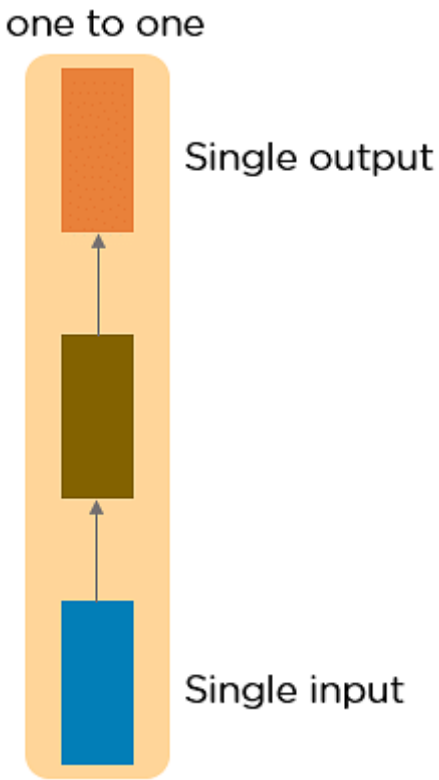
Types of Recurrent Neural Networks

There are four types of Recurrent Neural Networks:

1. One to One
2. One to Many
3. Many to One
4. Many to Many

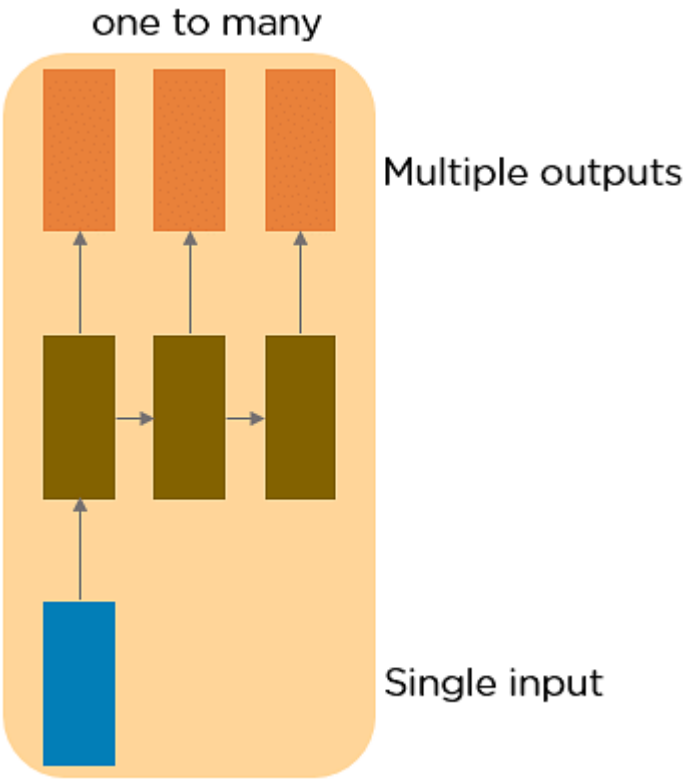
One to One RNN

This type of neural network is known as the Vanilla Neural Network. It's used for general machine learning problems, which has a single input and a single output.



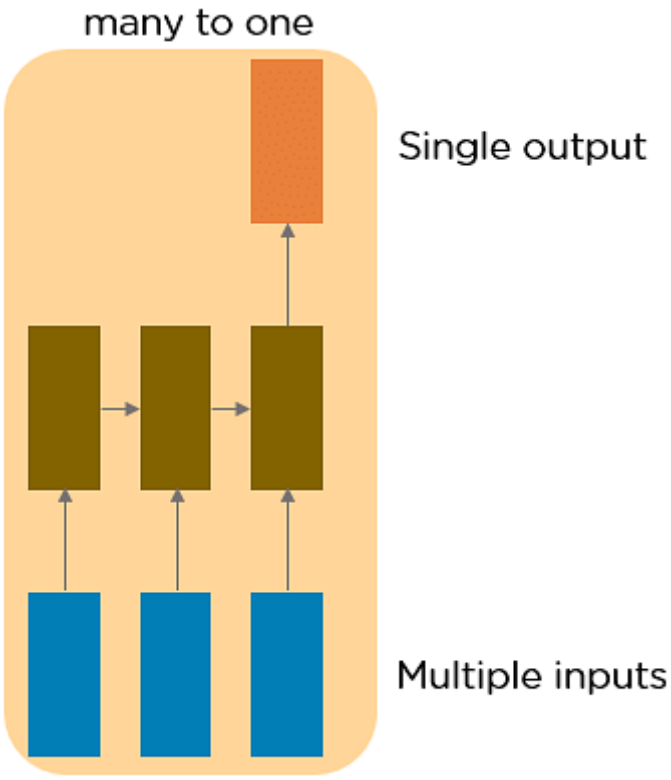
One to Many RNN

This type of neural network has a single input and multiple outputs. An example of this is the image caption.



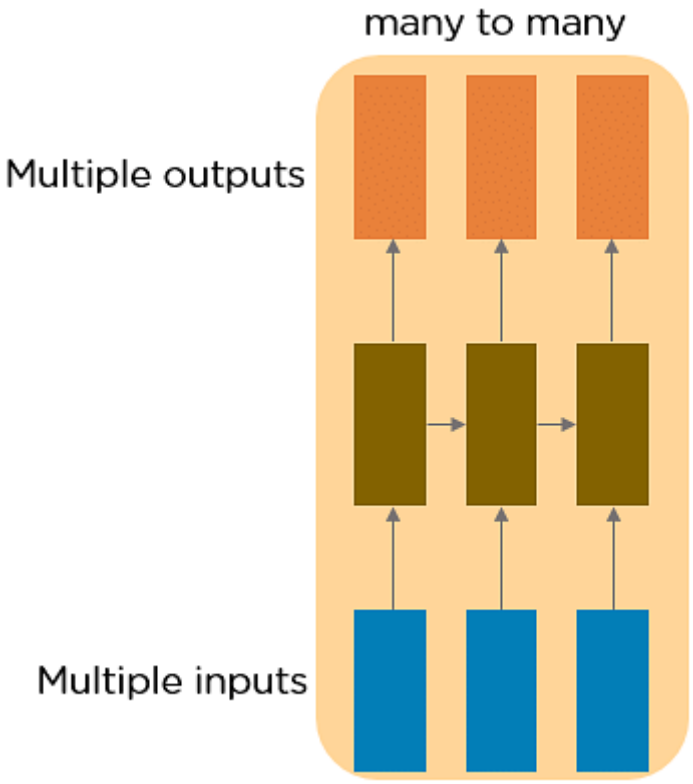
Many to One RNN

This RNN takes a sequence of inputs and generates a single output. Sentiment analysis is a good example of this kind of network where a given sentence can be classified as expressing positive or negative sentiments.



Many to Many RNN

This RNN takes a sequence of inputs and generates a sequence of outputs. Machine translation is one of the examples.

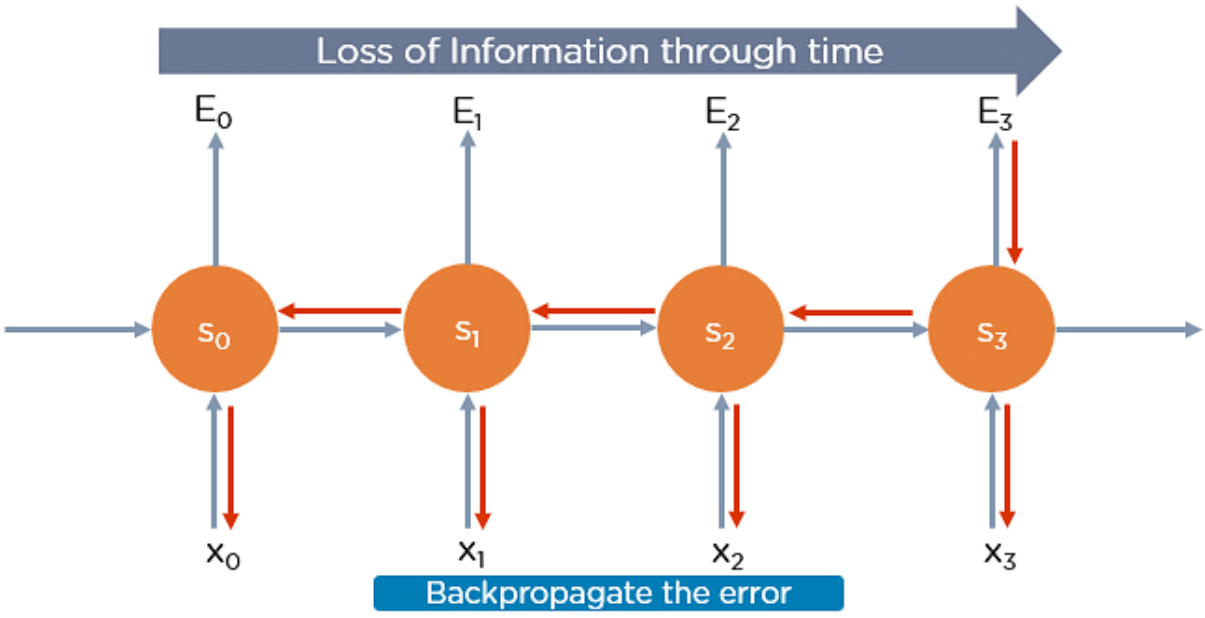


Two Issues of Standard RNNs

1. Vanishing Gradient Problem

Recurrent Neural Networks enable you to model time-dependent and sequential data problems, such as stock market prediction, machine translation, and text generation. You will find, however, RNN is hard to train because of the gradient problem.

RNNs suffer from the problem of vanishing gradients. The gradients carry information used in the RNN, and when the gradient becomes too small, the parameter updates become insignificant. This makes the learning of long data sequences difficult.

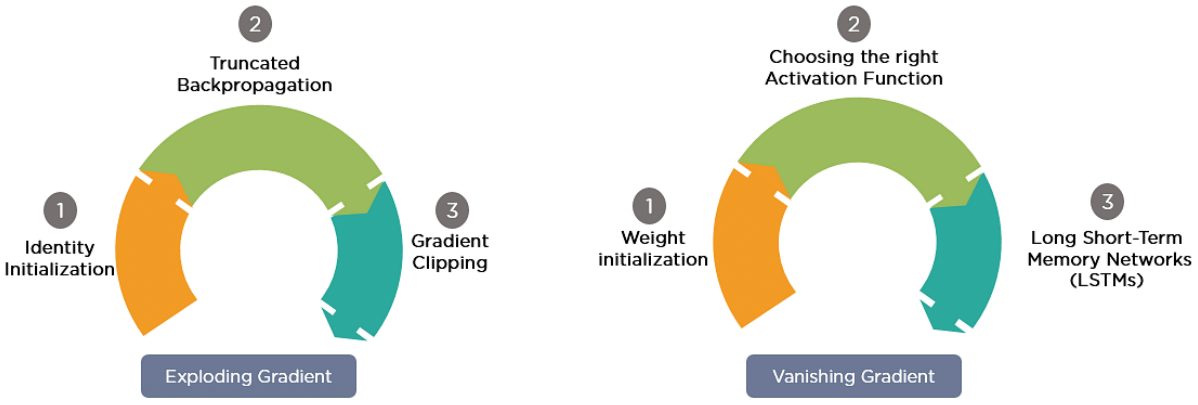


2. Exploding Gradient Problem

While training a neural network, if the slope tends to grow exponentially instead of decaying, this is called an Exploding Gradient. This problem arises when large error gradients accumulate, resulting in very large updates to the neural network model weights during the training process.

Long training time, poor performance, and bad accuracy are the major issues in gradient problems.

Gradient Problem Solutions



Now, let’s discuss the most popular and efficient way to deal with gradient problems, i.e., Long Short-Term Memory Network (LSTMs).

First, let’s understand Long-Term Dependencies.

Suppose you want to predict the last word in the text: “The clouds are in the _____.”

The most obvious answer to this is the “sky.” We do not need any further context to predict the last word in the above sentence.

Consider this sentence: “I have been staying in Spain for the last 10 years…I can speak fluent _____.”

The word you predict will depend on the previous few words in context. Here, you need the context of Spain to predict the last word in the text, and the most suitable answer to this sentence is “Spanish.” The gap between the relevant information and the point where it's needed may have become very large. LSTMs help you solve this problem.

Backpropagation Through Time

Backpropagation through time is when we apply a Backpropagation algorithm to a Recurrent Neural network that has time series data as its input.

In a typical RNN, one input is fed into the network at a time, and a single output is obtained. But in backpropagation, you use the current as well as the previous inputs as input. This is called a timestep and one timestep will consist of many time series data points entering the RNN simultaneously.

Once the neural network has trained on a timeset and given you an output, that output is used to calculate and accumulate the errors. After this, the network is rolled back up and weights are recalculated and updated keeping the errors in mind.

Long Short-Term Memory Networks

LSTMs are a special kind of RNN — capable of learning long-term dependencies by remembering information for long periods is the default behavior.

All RNN are in the form of a chain of repeating modules of a neural network. In standard RNNs, this repeating module will have a very simple structure, such as a single tanh layer.

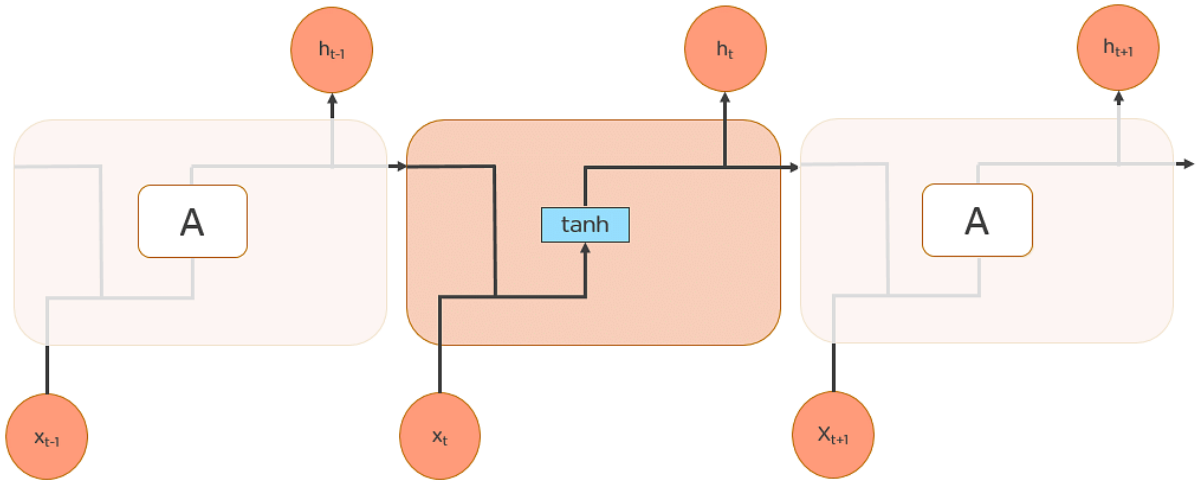
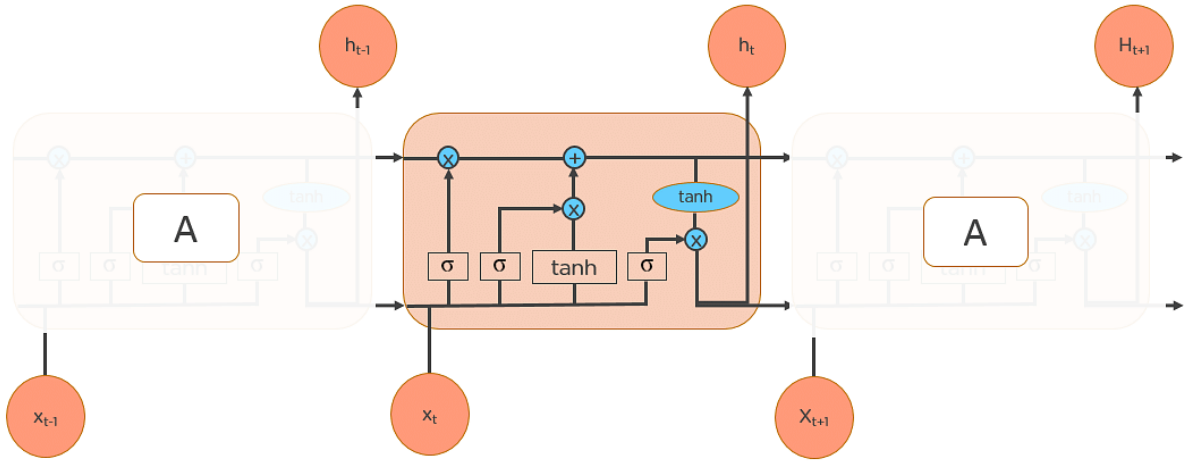
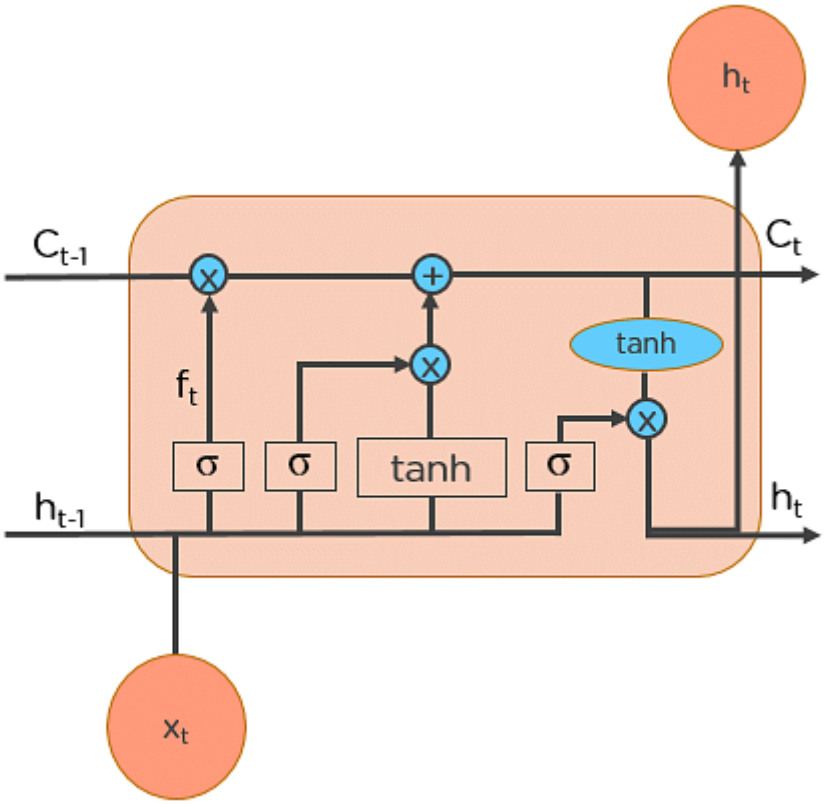


Fig: Long Short Term Memory Networks

LSTMs also have a chain-like structure, but the repeating module is a bit different structure. Instead of having a single neural network layer, four interacting layers are communicating extraordinarily.



Workings of LSTMs in RNN



LSTMs work in a 3-step process.

Step 1: Decide How Much Past Data It Should Remember

The first step in the LSTM is to decide which information should be omitted from the cell in that particular time step. The sigmoid function determines this. It looks at the previous state (ht-1) along with the current input xt and computes the function.

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

f_t = forget gate
Decides which information to delete that is not important from previous time step

Consider the following two sentences:

Let the output of h(t-1) be “Alice is good in Physics. John, on the other hand, is good at Chemistry.”

Let the current input at x(t) be “John plays football well. He told me yesterday over the phone that he had served as the captain of his college football team.”

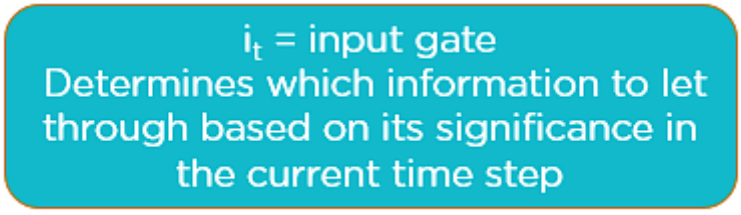
The forget gate realizes there might be a change in context after encountering the first full stop. It compares with the current input sentence at x(t). The next sentence talks about John, so the information on Alice is deleted. The position of the subject is vacated and assigned to John.

Step 2: Decide How Much This Unit Adds to the Current State

In the second layer, there are two parts. One is the sigmoid function, and the other is the tanh function. In the sigmoid function, it decides which values to let through (0 or 1). tanh function gives weightage to the values which are passed, deciding their level of importance (-1 to 1).

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

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



With the current input at x(t), the input gate analyzes the important information — John plays football, and the fact that he was the captain of his college team is important.

“He told me yesterday over the phone” is less important; hence it's forgotten. This process of adding some new information can be done via the input gate.

Step 3: Decide What Part of the Current Cell State Makes It to the Output

The third step is to decide what the output will be. First, we run a sigmoid layer, which decides what parts of the cell state make it to the output. Then, we put the cell state through tanh to push the values to be between -1 and 1 and multiply it by the output of the sigmoid gate.

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

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

o_t = output gate
Allows the passed in information to impact the output in the current time step

Let’s consider this example to predict the next word in the sentence: “John played tremendously well against the opponent and won for his team. For his contributions, brave ____ was awarded player of the match.”

There could be many choices for the empty space. The current input brave is an adjective, and adjectives describe a noun. So, “John” could be the best output after brave.

Use cases of RNN

1. Speech, text recognition & sentiment classification
2. Music synthesis
3. Image captioning

4. Chatbots & NLP
5. Machine Translation—Language translation
6. Stock predictions
7. Finding and blocking abusive words in a speech or text

Does RNN sound so powerful? No ! there is a big problem.

Problems with RNN :

Exploding and vanishing gradient problems during backpropagation.

Gradients are those values which to update neural networks weights. In other words, we can say that Gradient carries information.

Vanishing gradient is a big problem in deep neural networks. it vanishes or explodes quickly in earlier layers and this makes RNN unable to hold information of longer sequence. and thus **RNN becomes short-term memory**.

If we apply RNN for a paragraph RNN may leave out necessary information due to gradient problems and not be able to carry information from the initial time step to later time steps.

To solve this problem LSTM, GRU came into the picture.

How do LSTM, GRU solve this problem?

I highly encourage you to read Colah's blog for in-depth knowledge of LSTM.

The reason for exploding gradient was the capturing of relevant and irrelevant information. a model which can decide what information from a

paragraph and relevant and remember only relevant information and throw all the irrelevant information

This is achieved by using gates. the LSTM (Long -short-term memory) and GRU (Gated Recurrent Unit) have gates as an internal mechanism, which control what information to keep and what information to throw out. By doing this LSTM, GRU networks solve the exploding and vanishing gradient problem.

Almost each and every SOTA (state of the art) model based on RNN follows LSTM or GRU networks for prediction.

LSTMs /GRUs are implemented in speech recognition, text generation, caption generation, etc.

GRU Networks:

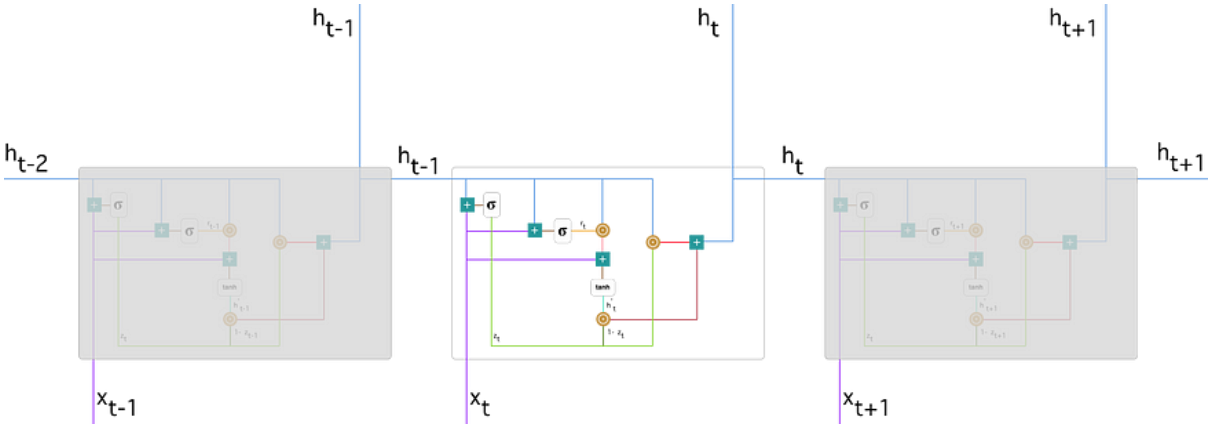
In this article, I will try to give a fairly simple and understandable explanation of one really fascinating type of neural network. Introduced by Cho, et al. in 2014, GRU (Gated Recurrent Unit) aims to solve the **vanishing gradient problem** which comes with a standard recurrent neural network. GRU can also be considered as a variation on the LSTM because both are designed similarly and, in some cases, produce equally excellent results. If you are not familiar with Recurrent Neural Networks, I recommend reading my brief introduction. For a better understanding of LSTM, many people recommend Christopher Olah's article. I would also add this paper which gives a clear distinction between GRU and LSTM.

How do GRUs work?

As mentioned above, GRUs are improved version of standard recurrent neural network. But what makes them so special and effective?

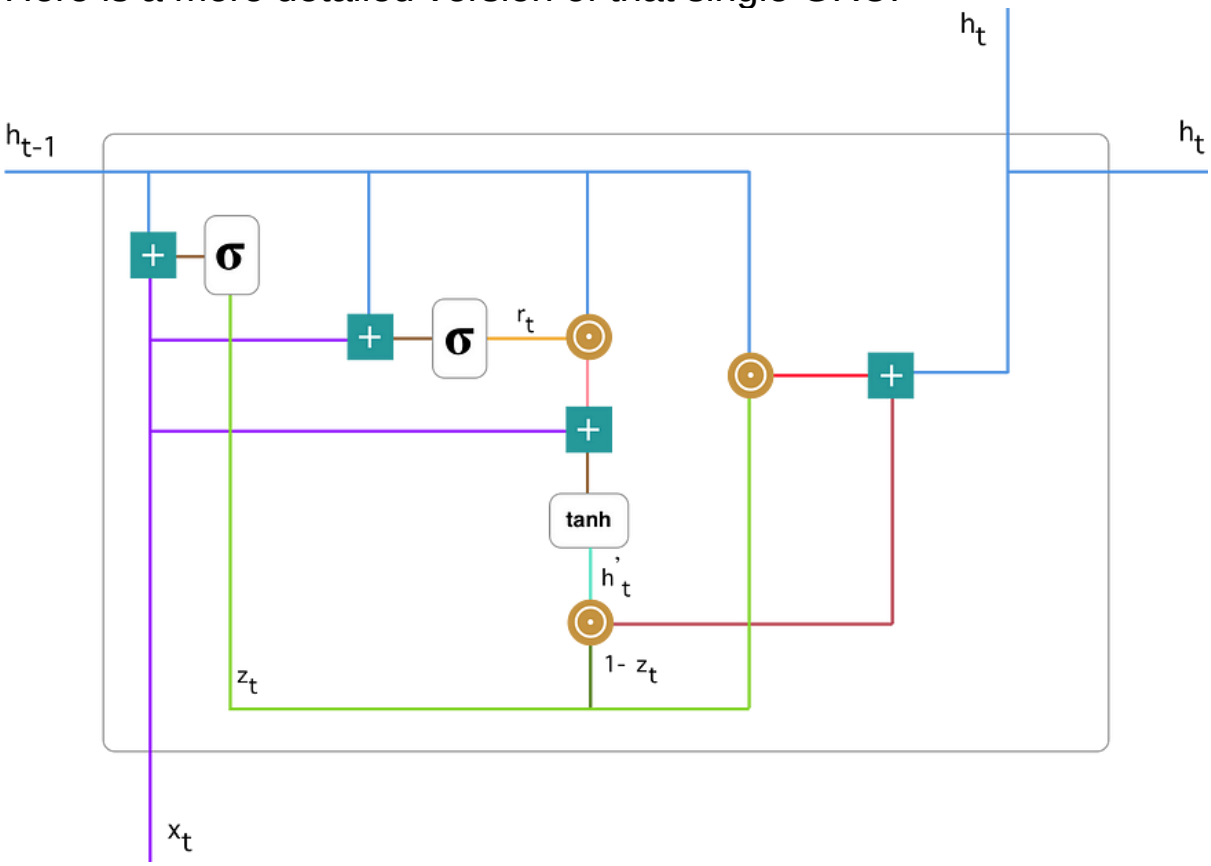
To solve the vanishing gradient problem of a standard RNN, GRU uses, so-called, **update gate and reset gate**. Basically, these are two vectors which decide what information should be passed to the output. The special thing about them is that they can be trained to keep information from long ago, without washing it through time or remove information which is irrelevant to the prediction.

To explain the mathematics behind that process we will examine a single unit from the following recurrent neural network:



Recurrent neural network with Gated Recurrent Unit

Here is a more detailed version of that single GRU:



Gated Recurrent Unit

First, let's introduce the notations:

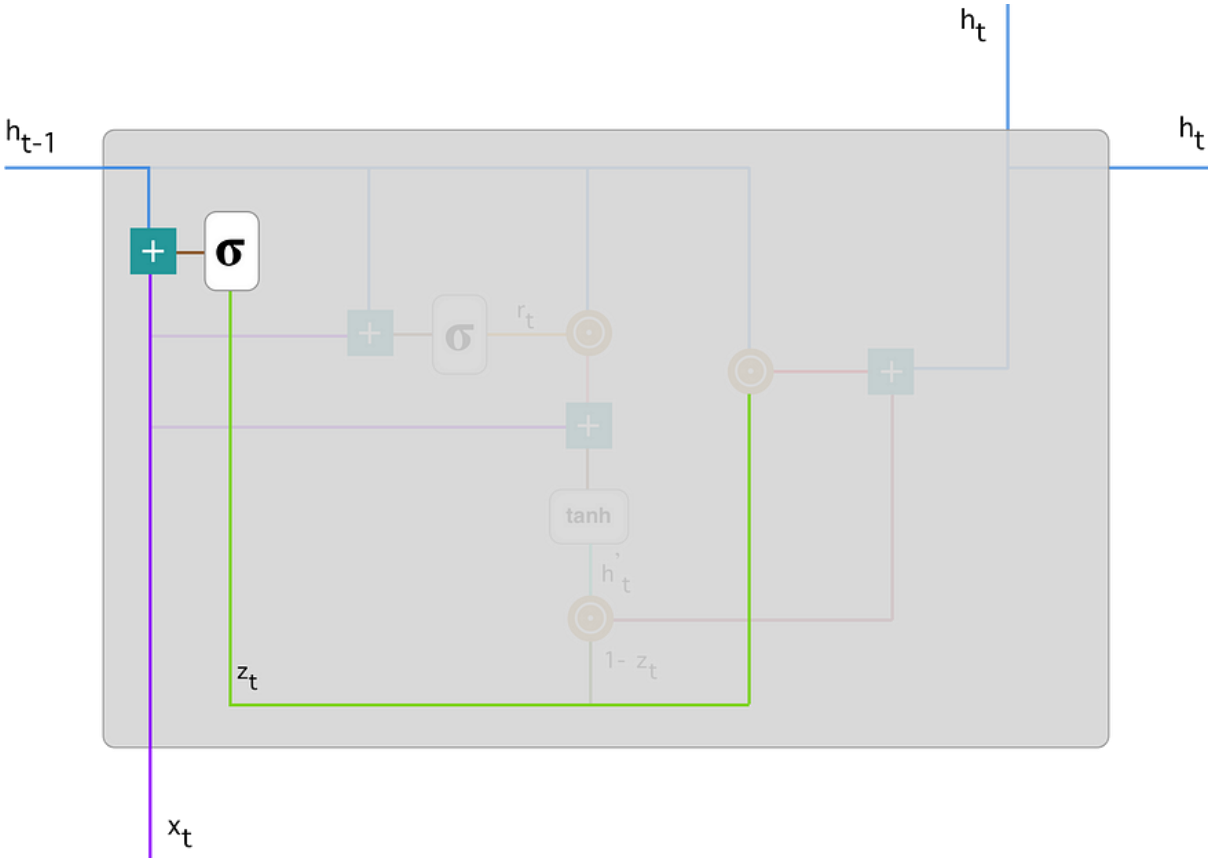
If you are not familiar with the above terminology, I recommend watching these tutorials about "sigmoid" and "tanh" function and "Hadamard product" operation.

#1. Update gate

We start with calculating the **update gate z_t for time step t** using the formula:

$$z_t = \sigma(W^{(z)}x_t + U^{(z)}h_{t-1})$$

When *x_t* is plugged into the network unit, it is multiplied by its own weight *W(z)*. The same goes for *h_(t-1)* which holds the information for the previous *t-1* units and is multiplied by its own weight *U(z).* Both results are added together and a sigmoid activation function is applied to squash the result between 0 and 1. Following the above schema, we have:



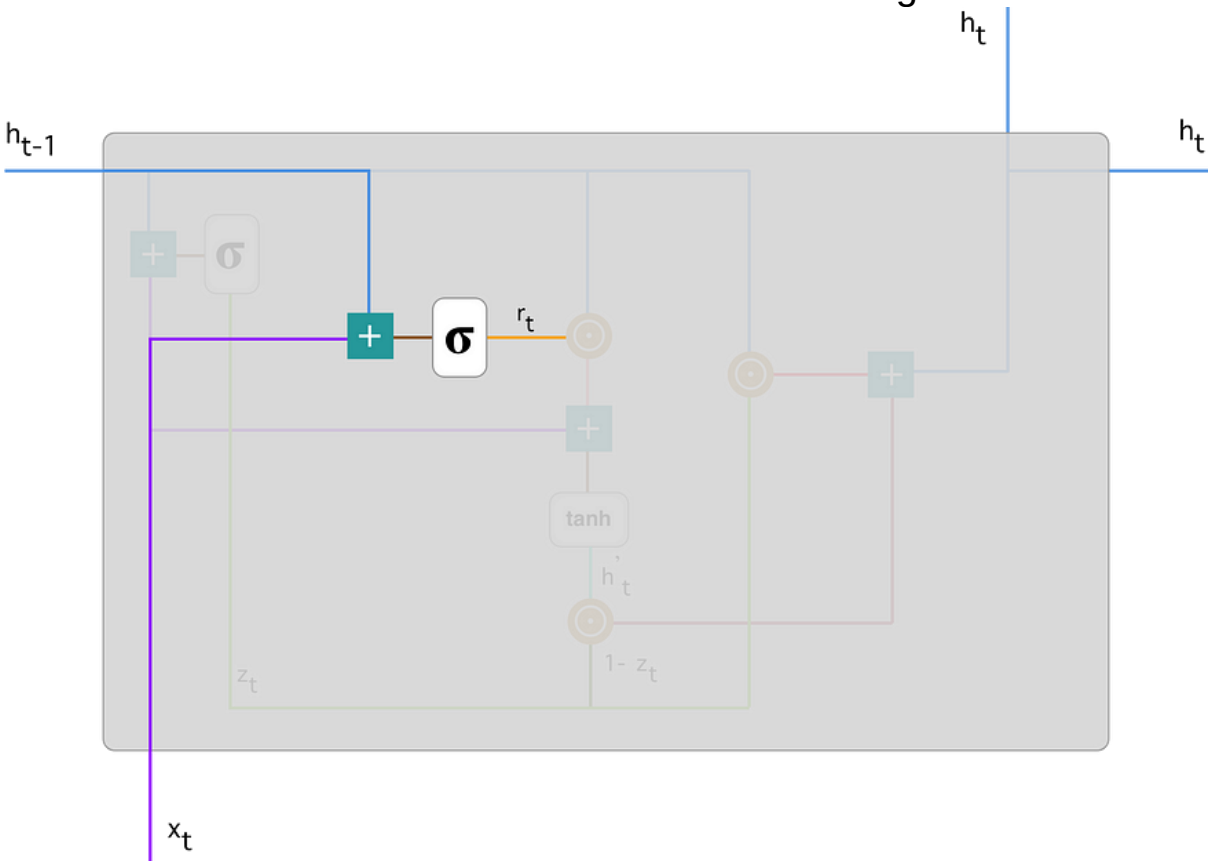
The update gate helps the model to determine how much of the past information (from previous time steps) needs to be passed along to the future. That is really powerful because the model can decide to copy all the information from the past and eliminate the risk of vanishing gradient problem. We will see the usage of the update gate later on. For now remember the formula for *z_t*.

#2. Reset gate

Essentially, **this gate is used from the model to decide how much of the past information to forget**. To calculate it, we use:

$$r_t = \sigma(W^{(r)}x_t + U^{(r)}h_{t-1})$$

This formula is the same as the one for the update gate. The difference comes in the weights and the gate's usage, which will see in a bit. The schema below shows where the reset gate is:



As before, we plug in *h_(t-1) — blue line* and *x_t — purple line*, multiply them with their corresponding weights, sum the results and apply the sigmoid function.

#3. Current memory content

Let's see how exactly the gates will affect the final output. First, we start with the usage of the reset gate. We introduce a new memory content which will use the reset gate to store the relevant information from the past. It is calculated as follows:

$$h_t' = \tanh(Wx_t + r_t \odot Uh_{t-1})$$

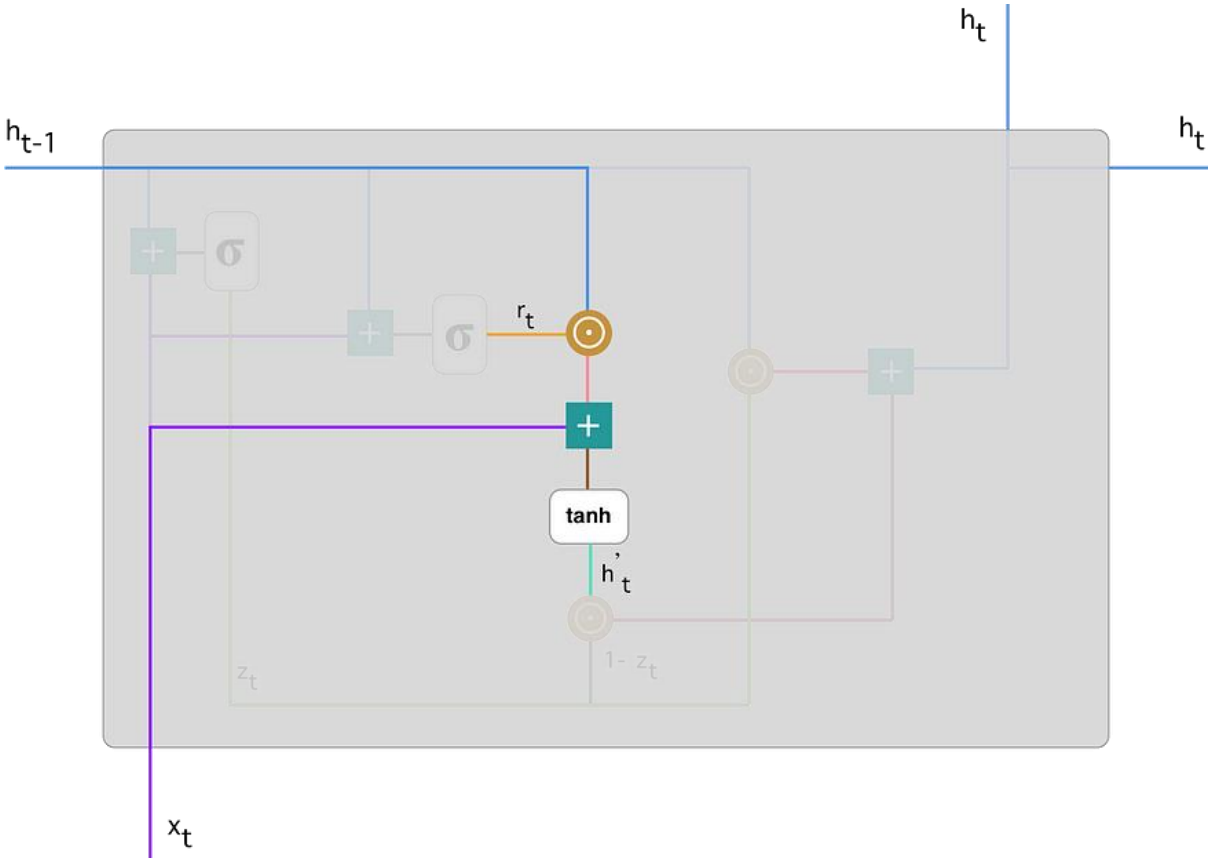
1. Multiply the input *x_t* with a weight *W* and *h_(t-1)* with a weight *U.*

2. Calculate the Hadamard (element-wise) product between the reset gate *r_t* and *Uh_(t-1).* That will determine what to remove from the previous time steps. Let's say we have a sentiment analysis problem for determining one's opinion about a book from a review he wrote. The text starts with "This is a fantasy book which illustrates…" and after a couple paragraphs ends with "I didn't quite enjoy the book because I think it captures too many details." To determine the overall level of satisfaction from the book we only need the last part of the review. In that case as the neural network approaches to the end of the text it will learn to assign *r_t* vector close to 0, washing out the past and focusing only on the last sentences.

3. Sum up the results of step 1 and 2.

4. Apply the nonlinear activation function *tanh*.

You can clearly see the steps here:



We do an element-wise multiplication of *h_(t-1) — blue line* and *r_t — orange line* and then sum the *result — pink line* with the input *x_t — purple line*. Finally, *tanh* is used to produce *h'_t — bright green line*.

#4. Final memory at current time step

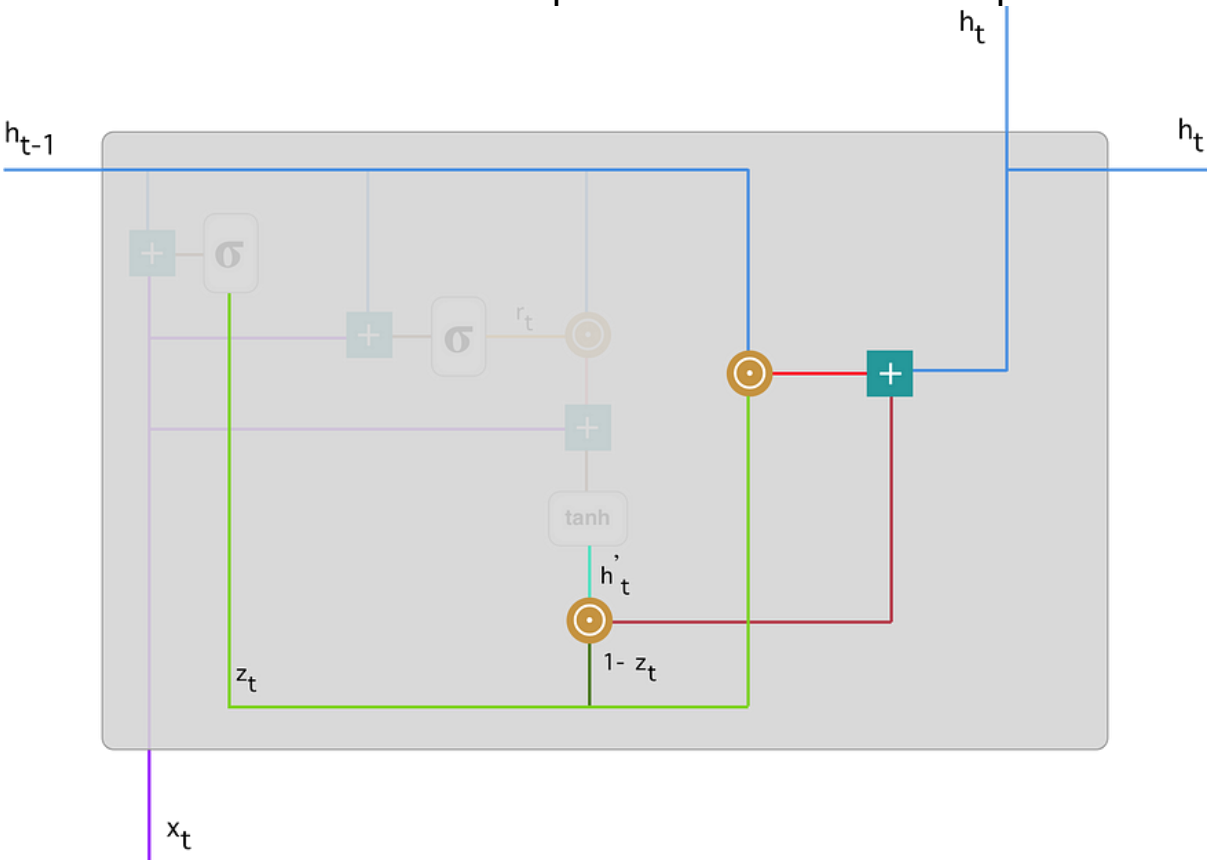
As the last step, the network needs to calculate *h_t* — vector which holds information for the current unit and passes it down to the network. In order to do that the update gate is needed. It determines what to collect from the current memory content — *h'_t* and what from the previous steps — *h_(t-1)*. That is done as follows:

$$h_t = z_t \odot h_{t-1} + (1 - z_t) \odot h_t^{'}$$

1. Apply element-wise multiplication to the update gate *z_t* and *h_(t-1).*
2. Apply element-wise multiplication to *(1-z_t)* and *h'_t.*
3. Sum the results from step 1 and 2.

Let's bring up the example about the book review. This time, the most relevant information is positioned in the beginning of the text. The model can learn to set the vector *z_t* close to 1 and keep a majority of the previous information. Since *z_t* will be close to 1 at this time step, *1-z_t* will be close to 0 which will ignore big portion of the current content (in this case the last part of the review which explains the book plot) which is irrelevant for our prediction.

Here is an illustration which emphasises on the above equation:



Following through, you can see how *z_t — green line* is used to calculate *1-z_t* which, combined with *h'_t — bright green line,* produces a result in the *dark red line*. *z_t* is also used with *h_(t-1) — blue line* in an element-wise multiplication. Finally, *h_t — blue line* is a result of the summation of the outputs corresponding to the *bright and dark red lines*.

Now, you can see how GRUs are able to store and filter the information using their update and reset gates. That eliminates the vanishing gradient problem since the model is not washing out the new input every single time but keeps the relevant information and passes it down to the next time steps of the network. **If carefully trained, they can perform extremely well even in complex scenarios.**

LSTM With Implementation

Introduction

Artificial Neural Networks (ANN) have paved a new path to the emerging AI industry since decades it has been introduced. With no doubt in its massive performance and architectures proposed over the decades, traditional machine-learning algorithms are on the verge of extinction with deep neural networks, in many real-world AI cases.

But, every new invention in technology must come with a drawback, otherwise, scientists cannot strive and discover something better to compensate for the previous drawbacks. Similarly, Neural Networks also came up with some loopholes that called for the invention of recurrent neural networks.

Why Recurrent?

In Neural Networks, we stack up various layers, composed of nodes that contain hidden layers, which are for learning and a dense layer for generating output. But, the central loophole in neural networks is that it does not have memory. Since no memory is associated, it becomes very difficult to work on sequential data like text corpora where we have sentences associated with each other, and even time-series where data is entirely sequential and dynamic.

Here, Recurrent Neural Networks comes to play. RNN addresses the memory issue by giving a feedback mechanism that looks back to the previous output and serves as a kind of memory. Since the previous outputs

gained during training leaves a footprint, it is very easy for the model to predict the future tokens (outputs) with help of previous ones.

Why LSTM when we have RNN?

A sentence or phrase only holds meaning when every word in it is associated with its previous word and the next one. LSTM, short for Long Short Term Memory, as 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, etc.

Tutorial Overview

In this tutorial, we will have an in-depth intuition about LSTM as well as see how it works with implementation!

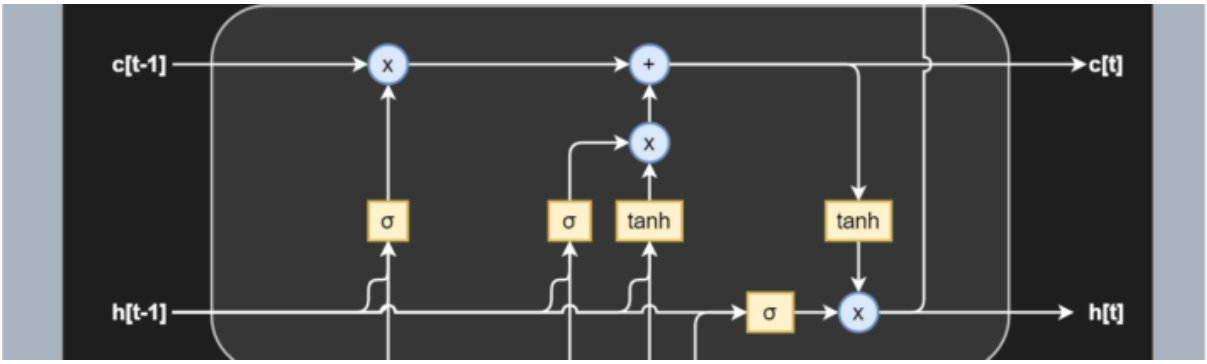
Let's have a look at what we will cover-

1. A Quick Look into LSTM Architecture
2. Why does LSTM outperform RNN?
3. *Deep Learning* about LSTM gates
4. An Implementation is Necessary!
5. Wrap Up with Bonus Resources

So, let's dive into the LSTM tutorial!

A Quick Look into LSTM Architecture

Let's see how a simple LSTM black box model looks-



Source -MachineCurve

To give a gentle introduction, LSTMs are nothing but a stack of neural networks composed of linear layers composed of weights and biases, just like any other standard neural network. The weights are constantly updated by backpropagation.

Now, before going in-depth, let me introduce a few crucial LSTM specific terms to you-

1. Cell—Every unit of the LSTM network is known as a “cell”. Each cell is composed of 3 inputs—

- *x(t)*—token at timestamp t
- *h(t*−1)—previous hidden state
- *c(t-1)*—previous cell state,

and 2 outputs—

- *h(t)*—updated hidden state, used for predicting the output
- *c(t)*—current cell state

2. Gates—LSTM uses a special theory of controlling the memorizing process. Popularly referred to as gating mechanism in LSTM, what the gates in LSTM do is, store the memory components in analog format, and make it

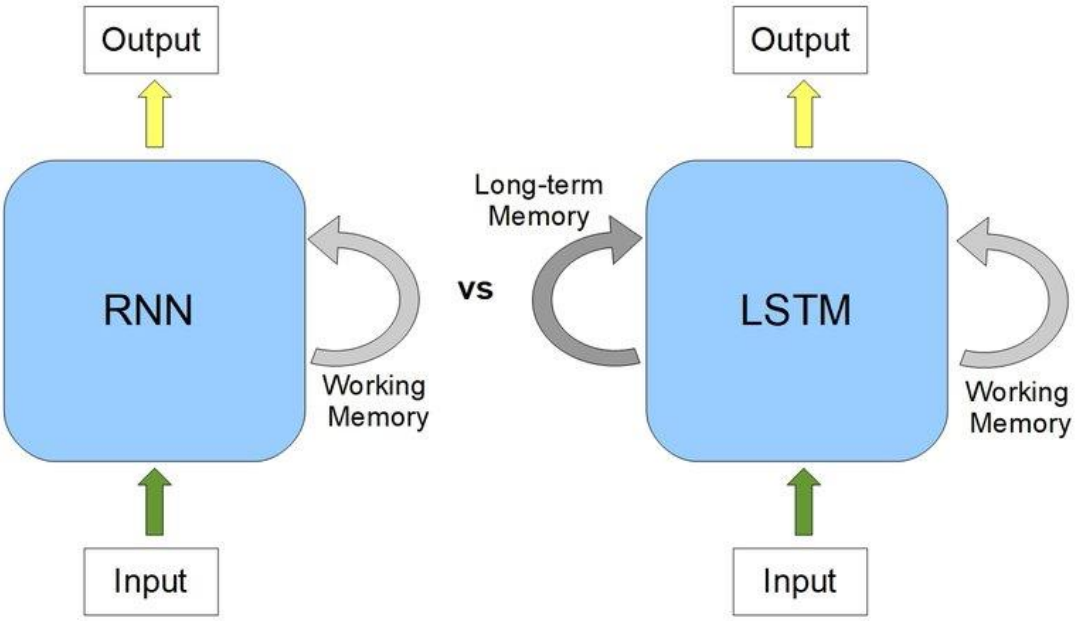
a probabilistic score by doing point-wise multiplication using sigmoid activation function, which stores it in the range of 0–1. Gates in LSTM regulate the flow of information in and out of the LSTM cells.

Gates are of 3 types—

- Input Gate—This gate lets in optional information necessary from the current cell state. It decides which information is relevant for the current input and allows it in.
- Output Gate—This gate updates and finalizes the next hidden state. Since the hidden state contains critical information about previous cell inputs, it decides for the last time which information it should carry for providing the output.
- Forget Gate—Pretty smart in eliminating unnecessary information, the forget gate multiplies 0 to the tokens which are not important or relevant and lets it be forgotten forever.

Why does LSTM outperform RNN?

First, let’s take a comparative look into an RNN and an LSTM-



Source – ResearchGate

RNNs have quite massively proved their incredible performance in sequence learning. But, it has been remarkably noticed that RNNs are not sporty while handling long-term dependencies.

Why so?

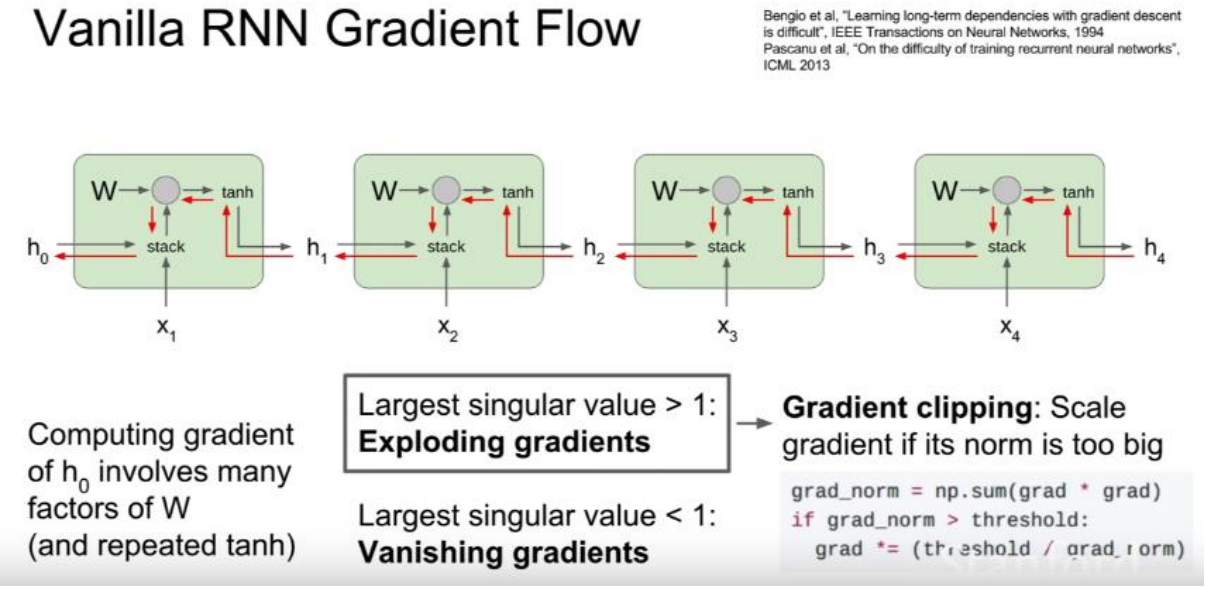
Vanishing Gradient Problem

Recurrent Neural Networks uses a hyperbolic tangent function, what we call the tanh function. The range of this activation function lies between [-1,1], with its derivative ranging from [0,1]. Now we know that RNNs are a deep sequential neural network. Hence, due to its depth, the matrix multiplications continually increase in the network as the input sequence keeps on increasing. Hence, while we use the chain rule of differentiation during calculating backpropagation, the network keeps on multiplying the numbers with small numbers. And guess what happens when you keep on multiplying a number with negative values with itself? It becomes exponentially smaller, squeezing the final gradient to almost 0, hence weights are no more updated,

and model training halts. It leads to poor learning, which we say as “cannot handle long term dependencies” when we speak about RNNs.

Exploding Gradient Problem

Similar concept to the vanishing gradient problem, but just the opposite of the process, let’s suppose in this case our gradient value is greater than 1 and multiplying a large number to itself makes it exponentially larger leading to the explosion of the gradient.



Source – Stanford NLP

This also leads to the major issue of Long Term Dependency.

Long Term Dependency Issue in RNNs

Let us consider a sentence-

“I am a data science student and I love machine _____.”

We know the blank has to be filled with ‘learning’. But had there been many terms after “I am a data science student” like, “I am a data science student pursuing MS from University of…… and I love machine _____”.

This time, however, RNNS fails to work. Likely in this case we do not need unnecessary information like “pursuing MS from University of……”. What LSTMs do is, leverage their forget gate to eliminate the unnecessary information, which helps them handle long-term dependencies.

Deep Learning about LSTM gates

Let’s learn about LSTM gates in-depth.

The Input Gate

As discussed earlier, the input gate optionally permits information that is relevant from the current cell state. It is the gate that determines which information is necessary for the current input and which isn’t by using the sigmoid activation function. It then stores the information in the current cell state. Next, comes to play the tanh activation mechanism, which computes the vector representations of the input-gate values, which are added to the cell state.

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

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

Source – Stanford NLP

The Forget Gate

We already discussed, while introducing gates, that the hidden state is responsible for predicting outputs. The output generated from the hidden state at (t-1) timestamp is h(t-1). After the forget gate receives the input x(t) and output from h(t-1), it performs a pointwise multiplication with its weight matrix with an add-on of sigmoid activation which generates probability scores. These probability scores help it determine what is useful information and what is irrelevant.

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

Source – Stanford NLP

Cell State Update Mechanism

Replacing the new cell state with whatever we had previously is not an LSTM thing! An LSTM, as opposed to an RNN, is clever enough to know that replacing the old cell state with new would lead to loss of crucial information required to predict the output sequence.

That’s where LSTM beats RNN, remember?!

By now, the input gate remembers which tokens are relevant and adds them to the current cell state with tanh activation enabled. Also, the forget gate output, when multiplied with the previous cell state C(t-1), discards the irrelevant information. Hence, combining these two gates’ jobs, our cell state is updated without any loss of relevant information or the addition of irrelevant ones.

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

Source – Stanford NLP

Output Gate

This gate, which pretty much clarifies from its name that it is about to give us the output, does a quite straightforward job. The output gate decides what to output from our current cell state. The output gate, also has a matrix where weights are stored and updated by backpropagation. This weight matrix, takes in the input token x(t) and the output from previously hidden state h(t-1) and does the same old pointwise multiplication task. However, as said earlier, this takes place on top of a sigmoid activation as we need probability scores to determine what will be the output sequence.

After we get the sigmoid scores, we simply multiply it with the updated cell-state, which contains some relevant information required for the final output prediction. A final tanh multiplication is applied at the very last, to ensure the values range from [-1,1], and our output sequence is ready!

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

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

An Implementation is Necessary!

Step 1: Import the dependencies and code the activation functions-

```
import random
import numpy as np
import math
def sigmoid(x):
```

```
    return 1. / (1 + np.exp(-x))
def sigmoid_derivative(values):
    return values*(1-values)
def tanh_derivative(values):
    return 1. - values ** 2
# createst uniform random array w/ values in [a,b) and shape args
def rand_arr(a, b, *args):
    np.random.seed(0)
    return np.random.rand(*args) * (b - a) + a
```

Step 2: Initializing the biases and weight matrices

```
class LstmParam:
    def __init__(self, mem_cell_ct, x_dim):
        self.mem_cell_ct = mem_cell_ct
        self.x_dim = x_dim
        concat_len = x_dim + mem_cell_ct
        # weight matrices
        self.wg = rand_arr(-0.1, 0.1, mem_cell_ct, concat_len)
        self.wi = rand_arr(-0.1, 0.1, mem_cell_ct, concat_len)
        self.wf = rand_arr(-0.1, 0.1, mem_cell_ct, concat_len)
        self.wo = rand_arr(-0.1, 0.1, mem_cell_ct, concat_len)
        # bias terms
        self.bg = rand_arr(-0.1, 0.1, mem_cell_ct)
        self.bi = rand_arr(-0.1, 0.1, mem_cell_ct)
        self.bf = rand_arr(-0.1, 0.1, mem_cell_ct)
        self.bo = rand_arr(-0.1, 0.1, mem_cell_ct)
        # diffs (derivative of loss function w.r.t. all parameters)
        self.wg_diff = np.zeros((mem_cell_ct, concat_len))
        self.wi_diff = np.zeros((mem_cell_ct, concat_len))
        self.wf_diff = np.zeros((mem_cell_ct, concat_len))
        self.wo_diff = np.zeros((mem_cell_ct, concat_len))
        self.bg_diff = np.zeros(mem_cell_ct)
        self.bi_diff = np.zeros(mem_cell_ct)
        self.bf_diff = np.zeros(mem_cell_ct)
        self.bo_diff = np.zeros(mem_cell_ct)
```

Step 3: Multiplying forget gate with last cell state to forget irrelevant tokens

```
#stacking x(present input xt) and h(t-1)
xc = np.hstack((x,  h_prev))
#dot product of Wf(forget weight matrix and xc +bias)
self.state.f = sigmoid(np.dot(self.param.wf, xc) + self.param.bf)
#finally multiplying forget_gate(self.state.f) with previous cell state(s_prev)
#to get present state.
self.state.s = self.state.g * self.state.i + s_prev * self.state.f
```

Step 4: Sigmoid Activation decides which values to take in and tanh transforms new tokens to vectors

```
#xc already calculated above
self.state.i = sigmoid(np.dot(self.param.wi, xc) + self.param.bi)
#C(t)
self.state.g = np.tanh(np.dot(self.param.wg, xc) + self.param.bg)
```

Step 5: Calculate the present cell state

```
#to calculate the present state
self.state.s = self.state.g * self.state.i + s_prev * self.state.f
```

Step 6: Calculate the output state

```
#to calculate the output state
self.state.o = sigmoid(np.dot(self.param.wo, xc) + self.param.bo)
#output state h
self.state.h = self.state.s * self.state.o
```

So, this is how a single node of LSTM works!

Conclusion

Long Short Term Memories are very efficient for solving use cases that involve lengthy textual data. It can range from speech synthesis, speech recognition to machine translation and text summarization. I suggest you solve these use-cases with LSTMs before jumping into more complex architectures like Attention Models.