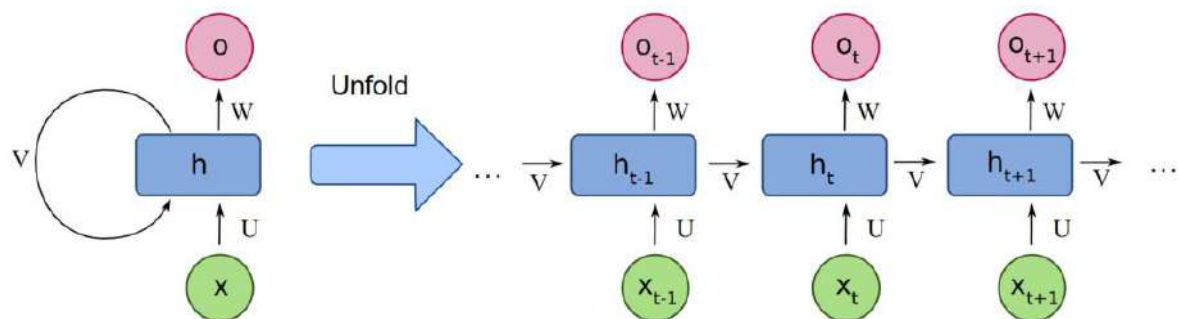


Introduction on Recurrent Neural Networks

A Deep Learning approach for modelling sequential data is **Recurrent Neural Networks (RNN)**. RNNs were the standard suggestion for working with sequential data before the advent of attention models. Specific parameters for each element of the sequence may be required by a deep feedforward model. It may also be unable to generalize to variable-length sequences.



Source: Medium.com

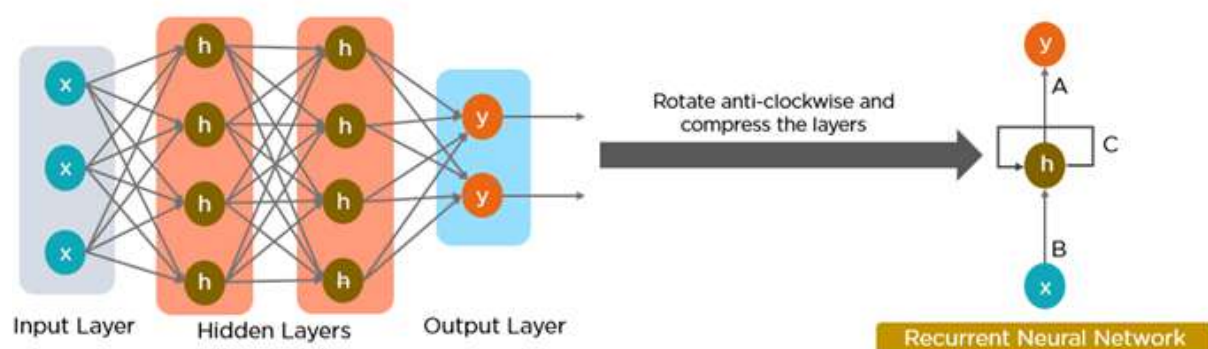
Recurrent Neural Networks use the same weights for each element of the sequence, decreasing the number of parameters and allowing the model to generalize to sequences of varying lengths. RNNs generalize to structured data other than sequential data, such as geographical or graphical data, because of its design.

Recurrent neural networks, like many other deep learning techniques, are relatively old. They were first developed in the 1980s, but we didn't appreciate their full potential until lately. The advent of long short-term memory (LSTM) in the 1990s, combined with an increase in computational power and the vast amounts of data that we now have to deal with, has really pushed RNNs to the forefront.

What is a Recurrent Neural Network (RNN)?

Neural networks imitate the function of the human brain in the fields of AI, machine learning, and deep learning, allowing computer programs to recognize patterns and solve common issues.

RNNs are a type of neural network that can be used to model sequence data. RNNs, which are formed from feedforward networks, are similar to human brains in their behaviour. Simply said, recurrent neural networks can anticipate sequential data in a way that other algorithms can't.



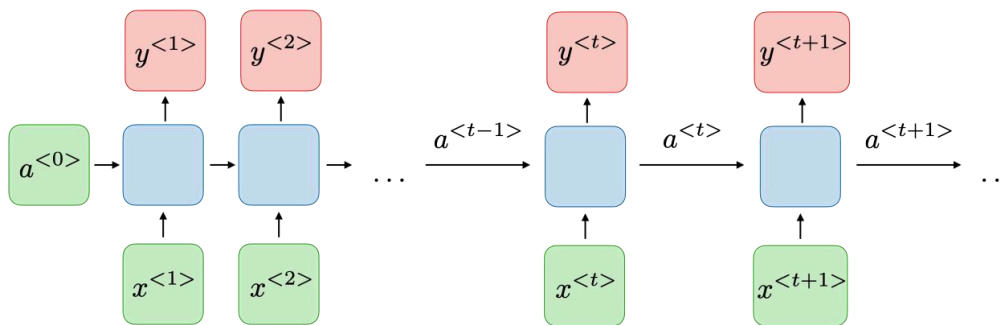
Source: Quora.com

All of the inputs and outputs in standard neural networks are independent of one another, however in some circumstances, such as when predicting the next word of a phrase, the prior words are necessary, and so the previous words must be remembered. As a result, RNN was created, which used a Hidden Layer to overcome the problem. The most important component of RNN is the Hidden state, which remembers specific information about a sequence.

RNNs have a Memory that stores all information about the calculations. It employs the same settings for each input since it produces the same outcome by performing the same task on all inputs or hidden layers.

The Architecture of a Traditional RNN

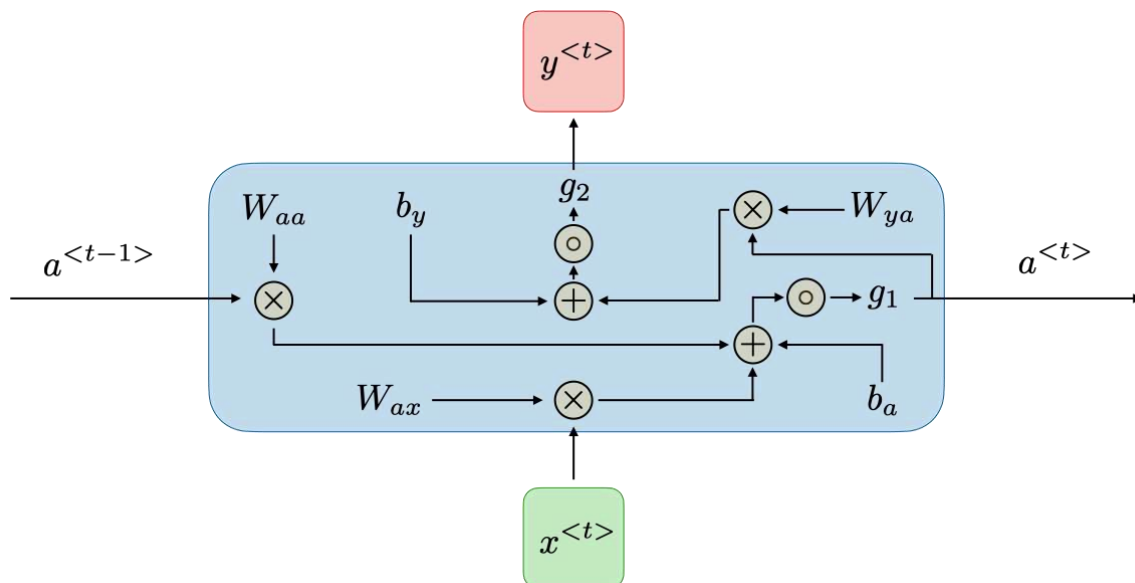
RNNs are a type of neural network that has hidden states and allows past outputs to be used as inputs. They usually go like this:



For each timestep t , the activation $a^{<t>}$ and the output $y^{<t>}$ are expressed as follows:

$$a^{<t>} = g_1(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a) \quad \text{and} \quad y^{<t>} = g_2(W_{ya}a^{<t>} + b_y)$$

where W_{ax} , W_{aa} , W_{ya} , b_a , b_y are coefficients that are shared temporally and g_1 , g_2 activation functions.



Source: Stanford.edu

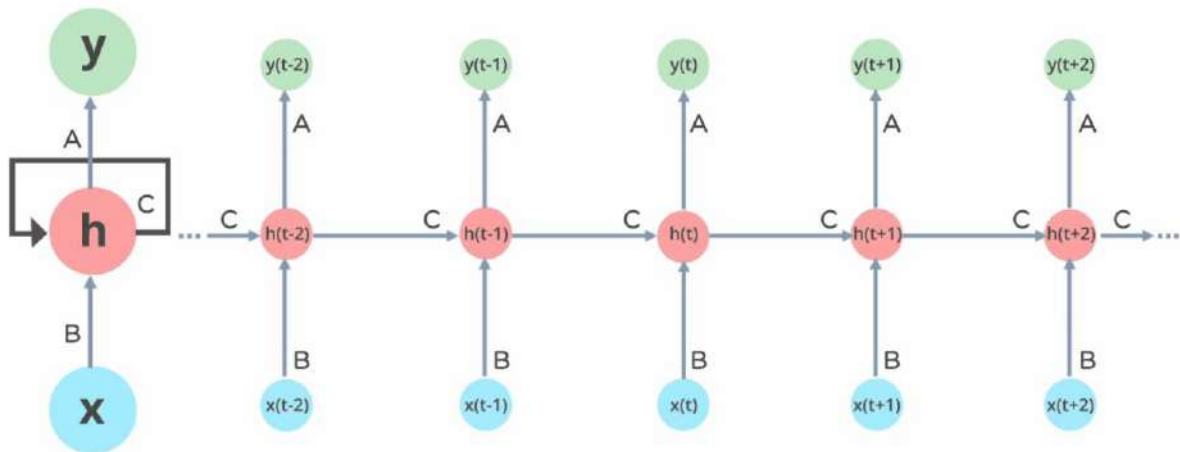
RNN architecture can vary depending on the problem you're trying to solve. From those with a single input and output to those with many (with variations between).

Below are some examples of RNN architectures that can help you better understand this.

- **One To One:** There is only one pair here. A one-to-one architecture is used in traditional neural networks.
- **One To Many:** A single input in a one-to-many network might result in numerous outputs. One too many networks are used in the production of music, for example.
- **Many To One:** In this scenario, a single output is produced by combining many inputs from distinct time steps. Sentiment analysis and emotion identification use such networks, in which the class label is determined by a sequence of words.
- **Many To Many:** For many to many, there are numerous options. Two inputs yield three outputs. Machine translation systems, such as English to French or vice versa translation systems, use many to many networks.

How does Recurrent Neural Networks work?

The information in recurrent neural networks cycles through a loop to the middle hidden layer.



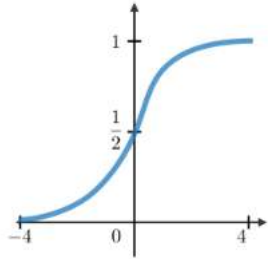
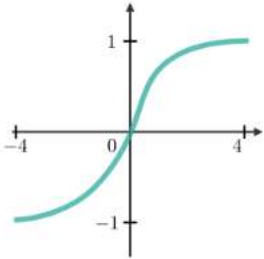
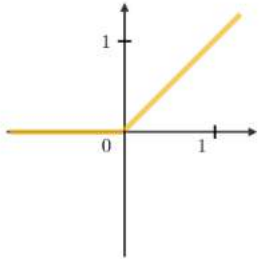
The input layer x receives and processes the neural network's input before passing it on to the middle layer.

Multiple hidden layers can be found in the middle layer h , each with its own activation functions, weights, and biases. You can utilize a recurrent neural network if the various parameters of different hidden layers are not impacted by the preceding layer, i.e. There is no memory in the neural network.

The different activation functions, weights, and biases will be standardized by the Recurrent Neural Network, ensuring that each hidden layer has the same characteristics. Rather than constructing numerous hidden layers, it will create only one and loop over it as many times as necessary.

Common Activation Functions

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

Sigmoid	Tanh	RELU
$g(z) = \frac{1}{1 + e^{-z}}$	$g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	$g(z) = \max(0, z)$
		

Source: MLtutorial.com

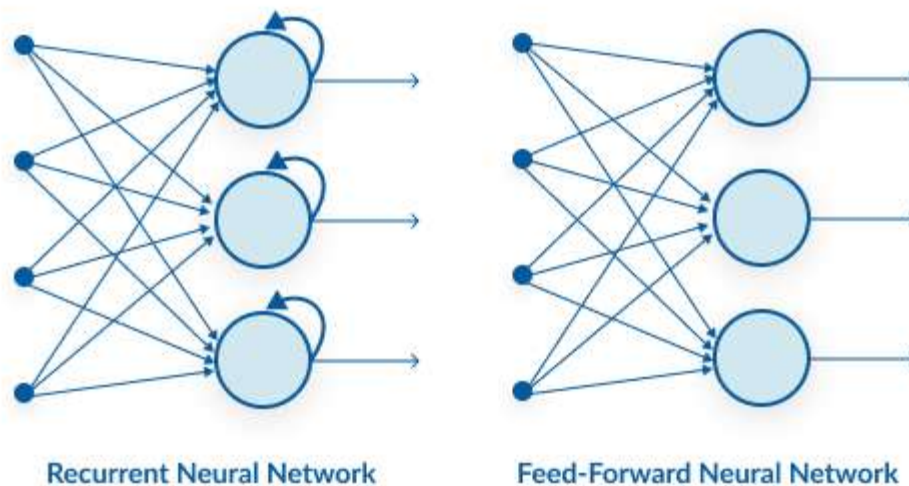
The following are some of the most commonly utilized functions:

- **Sigmoid:** The formula $g(z) = 1/(1 + e^{-z})$ is used to express this.
- **Tanh:** The formula $g(z) = (e^z - e^{-z})/(e^z + e^{-z})$ is used to express this.
- **Relu:** The formula $g(z) = \max(0, z)$ is used to express this.

Recurrent Neural Network Vs Feedforward Neural Network

A feed-forward neural network has only one route of information flow: from the input layer to the output layer, passing through the hidden layers. The data flows across the network in a straight route, never going through the same node twice.

The information flow between an RNN and a feed-forward neural network is depicted in the two figures below.



Source: Uditvani.com

Feed-forward neural networks are poor predictions of what will happen next because they have no memory of the information they receive. Because it simply analyses the current input, a feed-forward network has no idea of temporal order. Apart from its training, it has no memory of what transpired in the past.

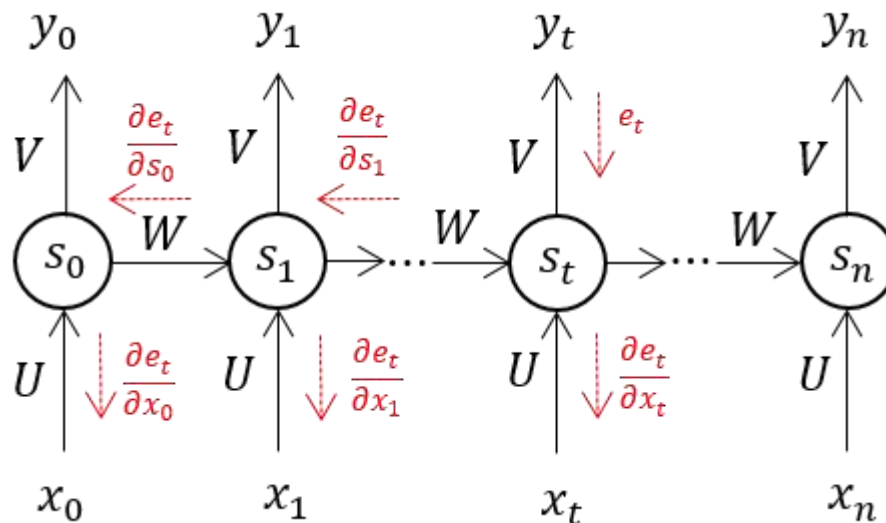
The information is in an RNN cycle via a loop. Before making a judgment, it evaluates the current input as well as what it has learned from past inputs. A recurrent neural network, on the other hand, may recall due to internal memory. It produces output, copies it, and then returns it to the network.

Backpropagation Through Time (BPTT)

When we apply a Backpropagation algorithm to a Recurrent Neural Network with time series data as its input, we call it backpropagation through time.

A single input is sent into the network at a time in a normal RNN, and a single output is obtained. Backpropagation, on the other hand, uses both the current and prior inputs as input. This is referred to as a timestep, and one

timestep will consist of multiple time series data points entering the RNN at the same time.

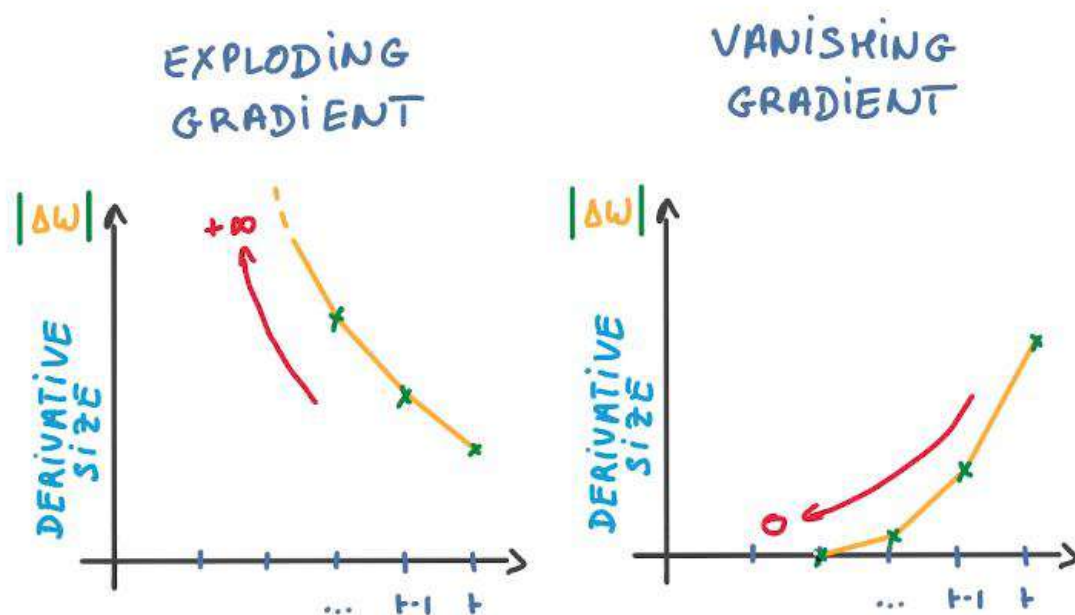


Source: Medium.com

The output of the neural network is used to calculate and collect the errors once it has trained on a time set and given you an output. The network is then rolled back up, and weights are recalculated and adjusted to account for the faults.

Two issues of Standard RNNs

There are two key challenges that RNNs have had to overcome, but in order to comprehend them, one must first grasp what a gradient is.



Source: GreatLearning.com

With regard to its inputs, a gradient is a partial derivative. If you're not sure what that implies, consider this: a gradient quantifies how much the output of a function varies when the inputs are changed slightly.

A function's slope is also known as its gradient. The steeper the slope, the faster a model can learn, the higher the gradient. The model, on the other hand, will stop learning if the slope is zero. A gradient is used to measure the change in all weights in relation to the change in error.

- **Exploding Gradients:** Exploding gradients occur when the algorithm gives the weights an absurdly high priority for no apparent reason. Fortunately, truncating or squashing the gradients is a simple solution to this problem.
- **Vanishing Gradients:** Vanishing gradients occur when the gradient values are too small, causing the model to stop learning or take far too long. This was a big issue in the 1990s, and it was far more difficult to

address than the exploding gradients. Fortunately, Sepp Hochreiter and Juergen Schmidhuber's LSTM concept solved the problem.

RNN Applications

Recurrent Neural Networks are used to tackle a variety of problems involving sequence data. There are many different types of sequence data, but the following are the most common: Audio, Text, Video, Biological sequences.

Using RNN models and sequence datasets, you may tackle a variety of problems, including :

- Speech recognition
- Generation of music
- Automated Translations
- Analysis of video action
- Sequence study of the genome and DNA

Basic Python Implementation (RNN with Keras)

Import the required libraries

```
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
```

Here's a simple Sequential model that processes integer sequences, embeds each integer into a 64-dimensional vector, and then uses an LSTM layer to handle the sequence of vectors.

```
model = keras.Sequential()
model.add(layers.Embedding(input_dim=1000, output_dim=64))
```

```
model.add(layers.LSTM(128))
model.add(layers.Dense(10))
model.summary()
```

Output:

Model: "sequential"

Layer (type)	Output Shape	Param #
=====		
=====		
embedding (Embedding)	(None, None, 64)	64000

lstm (LSTM)	(None, 128)	98816

dense (Dense)	(None, 10)	1290
=====		
=====		
Total params: 164,106		
Trainable params: 164,106		
Non-trainable params: 0		

2...

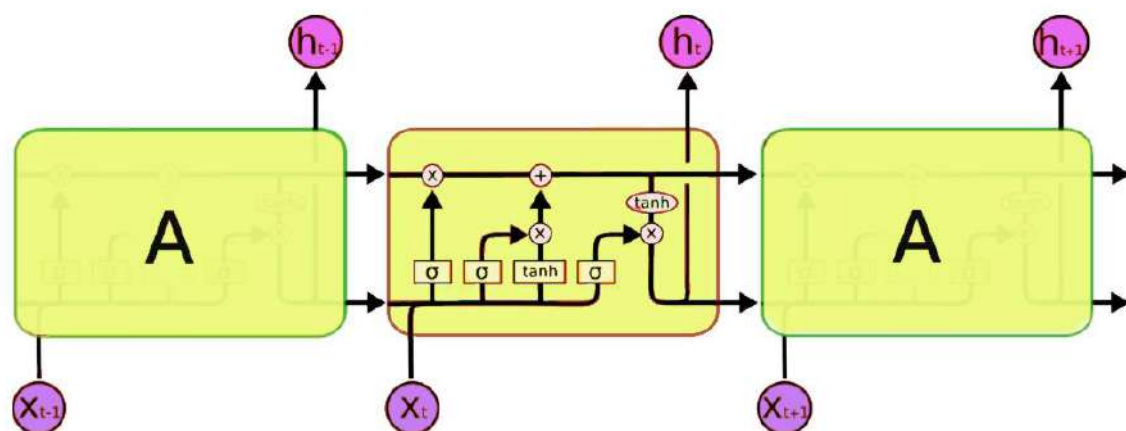
Architecture of LSTMs

The functioning of LSTM can be visualized by understanding the functioning of a news channel's team covering a murder story. Now, a news story is built around facts, evidence and statements of many people. Whenever a new event occurs you take either of the three steps.

You **input** this information into your news feed, right?

Now all these broken pieces of information cannot be served on mainstream media. So, after a certain time interval, you need to summarize this information and **output** the relevant things to your audience. Maybe in the form of *"XYZ turns out to be the prime suspect."*

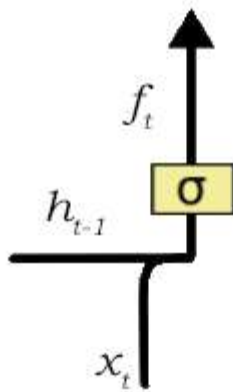
Now let's get into the details of the architecture of LSTM network:



Now, this is nowhere close to the simplified version which we saw before, but let me walk you through it. A typical LSTM network is comprised of different memory blocks called **cells** (the rectangles that we see in the image). There are two states that are being transferred to the next cell; the **cell state** and the **hidden state**. The memory blocks are responsible for remembering things and manipulations to this memory is done through three major mechanisms, called **gates**. Each of them is being discussed below.

4.1 Forget Gate

Taking the example of a text prediction problem. Let's assume an LSTM is fed in, the following sentence:



Bob is a nice person. Dan on the other hand is evil.

A forget gate is responsible for removing information from the cell state. The information that is no longer required for the LSTM to understand things or the information that is of less importance is removed via multiplication of a filter. This is required for optimizing the performance of the LSTM network.

This gate takes in two inputs; h_{t-1} and x_t .

h_{t-1} is the hidden state from the previous cell or the output of the previous cell and x_t is the input at that particular time step. The given inputs are multiplied by the weight matrices and a bias is added. The sigmoid function outputs a vector, with values ranging from 0 to 1, corresponding to each number in the cell state. Basically, the sigmoid function is responsible for deciding which values to keep and which to discard. If a '0' is output for a particular value in the cell state, it means that the forget gate wants the cell

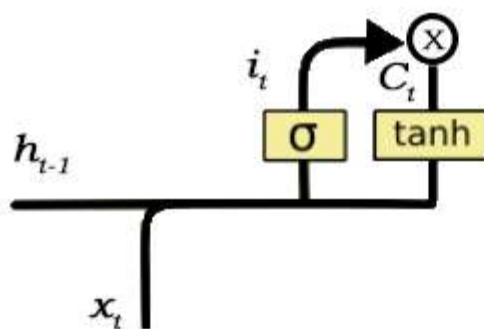
state to forget that piece of information completely. Similarly, a '1' means that the forget gate wants to remember that entire piece of information. This vector output from the sigmoid function is multiplied to the cell state.

Input Gate

Okay, let's take another example where the LSTM is analyzing a sentence:

Bob knows swimming. He told me over the phone that he had served the navy for 4 long years.

Now the important information here is that "Bob" knows swimming and that he has served the Navy for four years. This can be added to the cell state, however, the fact that he told all this over the phone is a less important fact and can be ignored. This process of adding some new information can be done via the **input** gate.



The input gate is responsible for the addition of information to the cell state. This addition of information is basically three-step process as seen from the diagram above.

1. Regulating what values need to be added to the cell state by involving a sigmoid function. This is basically very similar to the forget gate and acts as a filter for all the information from h_{t-1} and x_t .
2. Creating a vector containing all possible values that can be added (as perceived from h_{t-1} and x_t) to the cell state. This is done using the **tanh** function, which outputs values from -1 to +1.
3. Multiplying the value of the regulatory filter (the sigmoid gate) to the created vector (the tanh function) and then adding this useful information to the cell state via addition operation.

Once this three-step process is done with, we ensure that only that information is added to the cell state that is *important* and is not *redundant*.

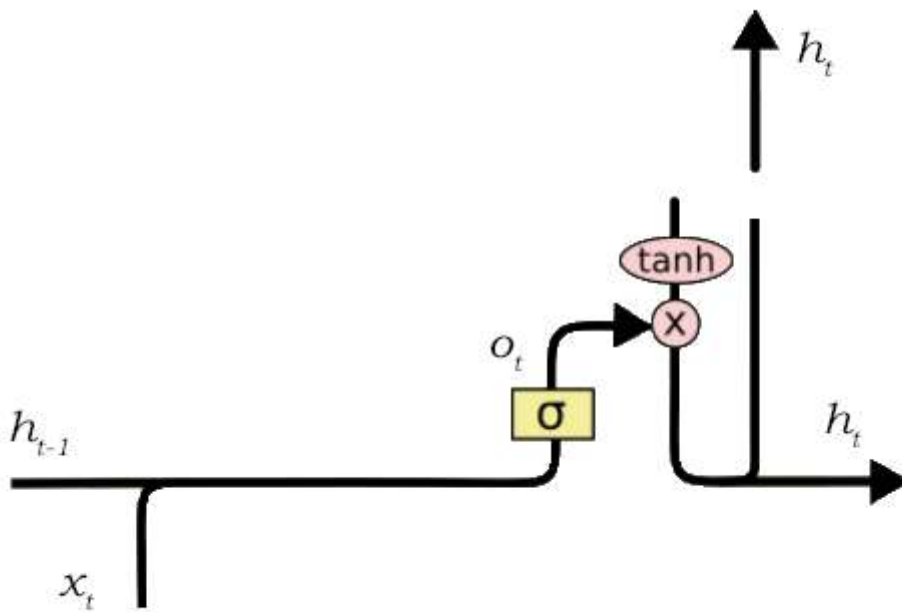
Output Gate

Not all information that runs along the cell state, is fit for being output at a certain time. We'll visualize this with an example:

Bob fought single handedly with the enemy and died for his country. For his contributions brave ____.

In this phrase, there could be a number of options for the empty space. But we know that the current input of '*brave*', is an adjective that is used to describe a noun. Thus, whatever word follows, has a strong tendency of being a noun. And thus, Bob could be an apt output.

This job of selecting useful information from the current cell state and showing it out as an output is done via the output gate. Here is its structure:



The functioning of an output gate can again be broken down to three steps:

1. Creating a vector after applying **tanh** function to the cell state, thereby scaling the values to the range -1 to +1.
2. Making a filter using the values of h_{t-1} and x_t , such that it can regulate the values that need to be output from the vector created above. This filter again employs a sigmoid function.
3. Multiplying the value of this regulatory filter to the vector created in step 1, and sending it out as a output and also to the hidden state of the next cell.

Limitations of lstm

Computationally expensive and memory-intensive.

Limited context window size and interpretability.

Applications of lstm

Google Translate uses LSTMs to translate text from one language to another.

Siri and Alexa use LSTMs to understand spoken language and respond to user requests.

Netflix uses LSTMs to recommend movies and TV shows to users.

Tesla uses LSTMs to train its self-driving cars.

Python code for lstm

```
import numpy as np
```

```
import keras.backend as K
```

```
from keras.models import Sequential
```

```
from keras.layers import LSTM, Dense
```

```
data = np.loadtxt('data.csv', delimiter=',')
```

```
x_train = data[:, :-1]
```

```
y_train = data[:, -1]
```

```
model = Sequential()
```

```
model.add(LSTM(128, input_shape=(x_train.shape[1],)))
```

```
model.add(Dense(1, activation='sigmoid'))
```

```
model.compile(optimizer='adam',                loss='binary_crossentropy',
metrics=['accuracy'])

model.fit(x_train, y_train, epochs=10)

score = model.evaluate(x_train, y_train)

print('Accuracy:', score[1])

predictions = model.predict(x_train)
```

3.Architecture of BiLSTM

The architecture of a Bidirectional Long Short-Term Memory (Bi-LSTM) neural network involves using two LSTM layers, one processing the input sequence in the forward direction, and the other processing it in the backward direction. Here is an overview of the architecture:

Input Layer: The input to a Bi-LSTM network is typically a sequence of data, which could be a sequence of words in natural language processing (NLP) tasks, time series data, or any other kind of sequential data. Each element in the sequence is represented as a feature vector.

Forward LSTM Layer: This LSTM layer processes the input sequence from left to right (i.e., forward in time). It consists of the following components:

Input Gate: Determines what new information should be added to the cell state.

Forget Gate: Decides what information from the previous state should be discarded.

Cell State: Stores information over time.

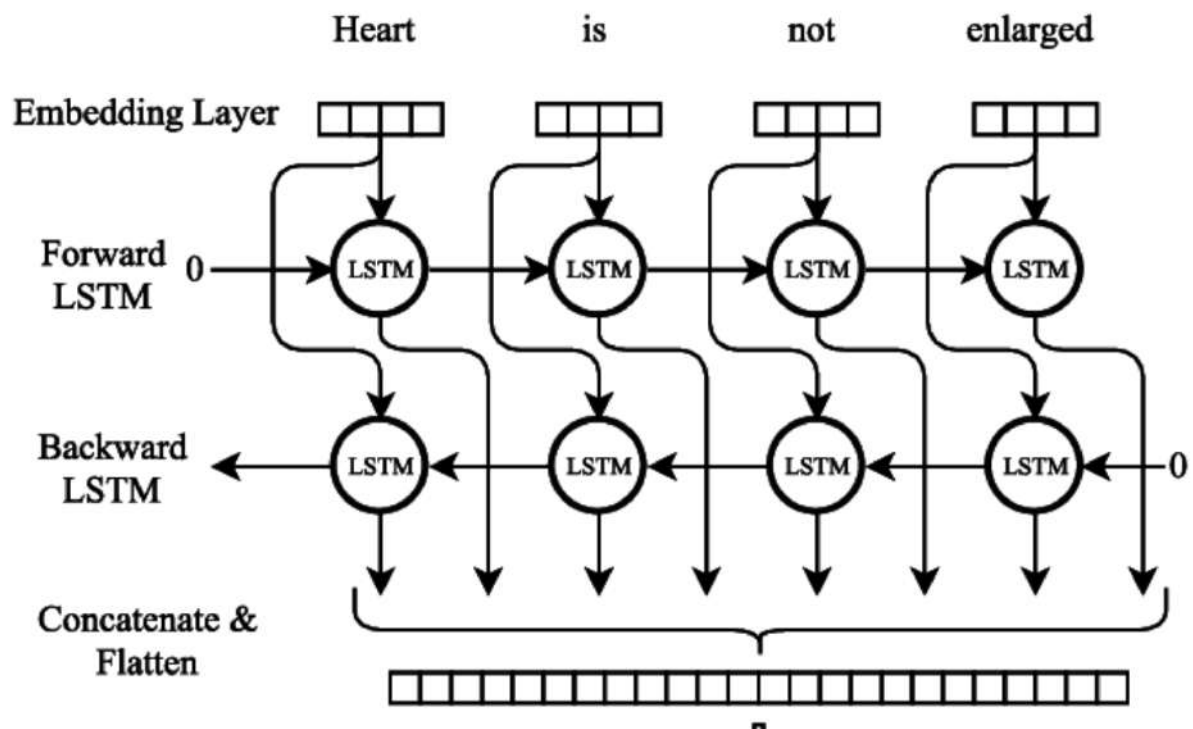
Output Gate: Controls what information should be output to the next layer.

Backward LSTM Layer: This LSTM layer processes the input sequence from right to left (i.e., backward in time) and has the same components as the forward LSTM layer.

Concatenation: The outputs from both the forward and backward LSTM layers are concatenated at each time step. This results in a combined representation for each time step that contains information from both the past and future context.

Optional Additional Layers: Depending on the specific task, additional layers like fully connected (dense) layers or other recurrent layers can be added on top of the concatenated output to further process the information.

Output Layer: The final layer of the network produces the desired output based on the task at hand. For example, in NLP tasks, it might be a softmax layer for classification or a regression layer for regression tasks.



Limitations of bilstm

Computationally expensive and memory-intensive.

Limited interpretability and may not be appropriate for all types of data.

Applications of bilstm

Machine translation: BiLSTM is used to translate text from one language to another.

Text summarization: BiLSTM is used to generate a summary of a long text passage.

Sentiment analysis: BiLSTM is used to identify the sentiment of a text passage, such as whether it is positive, negative, or neutral.

Question answering: BiLSTM is used to answer questions about a text passage.

Python code for bilstm

```
import numpy as np

from keras.preprocessing import sequence

from keras.models import Sequential

from keras.layers import Dense, Dropout, Embedding, LSTM, Bidirectional

(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=10000)

x_train = sequence.pad_sequences(x_train, maxlen=200)

x_test = sequence.pad_sequences(x_test, maxlen=200)

model = Sequential()

model.add(Embedding(10000, 128, input_length=200))

model.add(Bidirectional(LSTM(128)))

model.add(Dense(128, activation='relu'))

model.add(Dense(1, activation='sigmoid'))

model.compile(loss='binary_crossentropy', optimizer='adam',
metrics=['accuracy'])

model.fit(x_train, y_train, epochs=10)

score = model.evaluate(x_test, y_test)

print('Test accuracy:', score[1])
```

4. working of Gated recurrent unit

GRU, which stands for Gated Recurrent Unit, is a type of recurrent neural network (RNN) architecture commonly used in deep learning for sequential data processing tasks. GRUs are similar to Long Short-Term Memory (LSTM) networks in that they are designed to address the vanishing gradient problem and capture long-range dependencies in sequential data. However, GRUs are somewhat simpler in terms of architecture compared to LSTMs.

The architecture of a Gated Recurrent Unit (GRU) consists of several components, including gates and hidden states, which allow it to process sequential data efficiently. Here's an overview of the key components and their functions in a GRU:

Hidden State (h):

The hidden state at each time step captures the information that the GRU has learned from the input sequence up to that point.

It is analogous to the memory cell in an LSTM but is computed differently in the case of GRUs.

Input (x):

The input at each time step represents the current element in the input sequence.

It can be thought of as the feature vector or observation at that time step.

Reset Gate (r):

The reset gate is responsible for determining how much of the previous hidden state (h) should be "forgotten" or reset at the current time step.

It takes both the previous hidden state (h) and the current input (x) as inputs.

Mathematically, the reset gate is computed as a sigmoid function of the weighted sum of h and x .

Update Gate (z):

The update gate controls how much of the new candidate state should be added to the current hidden state.

It also takes the previous hidden state (h) and the current input (x) as inputs.

The update gate is computed as a sigmoid function of the weighted sum of h and x .

Candidate State (\tilde{h}):

The candidate state represents the new information from the current input (x) that could be added to the hidden state.

It is computed based on the reset gate (r) and the current input (x).

Current Hidden State (h_t):

The current hidden state (h_t) is computed as a combination of the previous hidden state (h_{t-1}) weighted by the update gate (z) and the candidate state (\tilde{h}) weighted by $(1 - z)$.

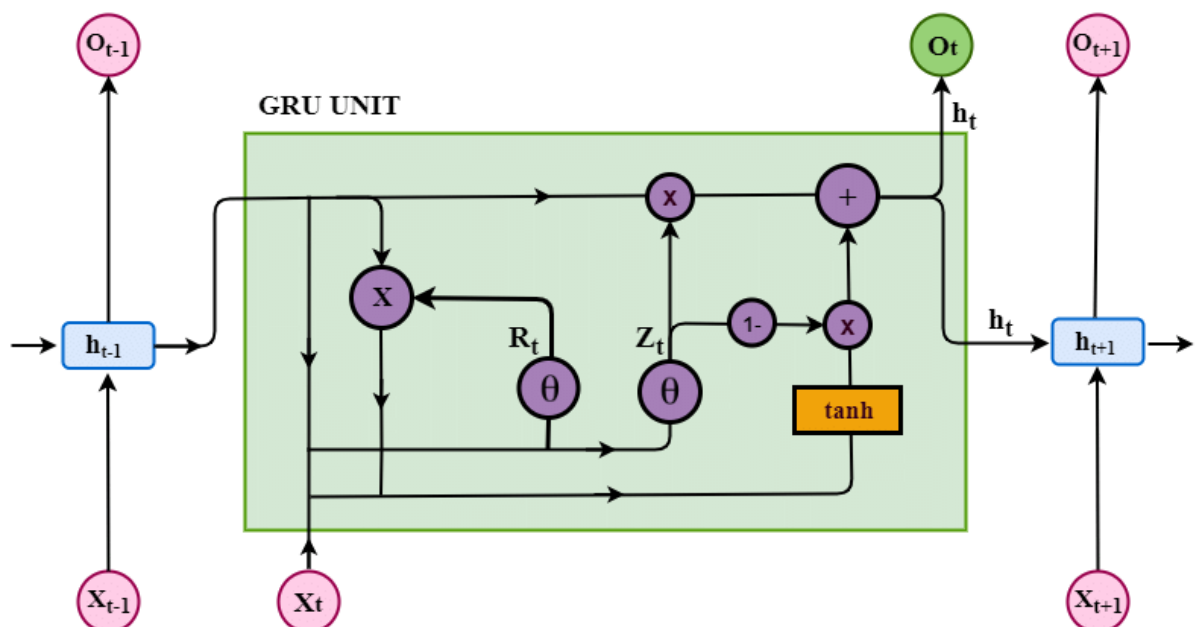
Mathematically, the equations that govern the behavior of a GRU at each time step are as follows:

Update Gate (z): $z_t = \text{sigmoid}(W_z * [h_{t-1}, x_t])$

Reset Gate (r): $r_t = \text{sigmoid}(W_r * [h_{t-1}, x_t])$

Candidate State (\tilde{h}): $\tilde{h}_t = \tanh(W_h * [r_t * h_{t-1}, x_t])$

Current Hidden State (h_t): $h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$



Limited Modeling Capacity: GRUs are less expressive compared to Long Short-Term Memory (LSTM) networks. LSTMs have an extra cell state, which allows them to capture longer-term dependencies in sequences. In cases where very long-range dependencies are critical, LSTMs might outperform GRUs.

Difficulty with Long Sequences: Similar to standard recurrent neural networks (RNNs), GRUs can still suffer from vanishing gradients when

dealing with very long sequences. This can hinder their ability to capture information from distant time steps effectively.

Complex Patterns: GRUs may struggle with tasks that require capturing complex temporal patterns. For instance, tasks that involve fine-grained timing or intricate dependencies in the data may benefit from more complex architectures like LSTMs.

Lack of Explicit Memory Cells: While GRUs have a hidden state that can retain information over time, they don't have a separate memory cell like LSTMs.

Python code for gru

```
import tensorflow as tf

class GRU(tf.keras.Model):

    def __init__(self, units, input_shape):

        super(GRU, self).__init__()

        self.gru = tf.keras.layers.GRU(units, input_shape=input_shape)

    def call(self, inputs):

        outputs = self.gru(inputs)

        return outputs

model = GRU(128, input_shape=(None, 10))
```

```
model.compile(optimizer='adam', loss='mse')
```

```
model.fit(x_train, y_train, epochs=10)
```

```
predictions = model.predict(x_test)
```