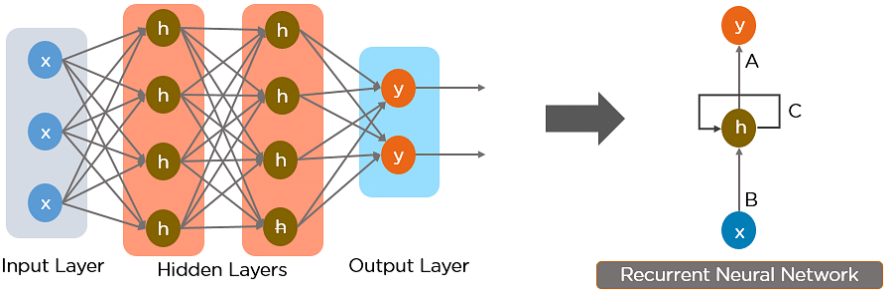
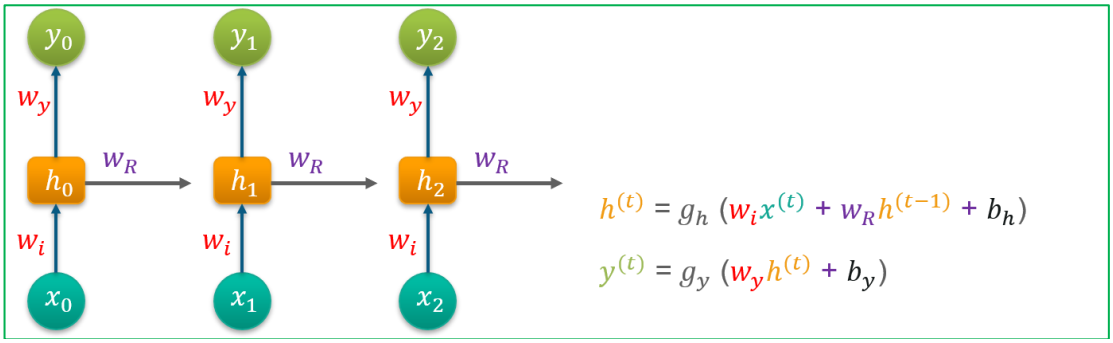
Recurrent and Recursive Networks: Recurrent Neural Networks, Bidirectional RNNs, Deep recurrent neural networks, Long Short-Term Memory Networks, and building an RNN for text classification.

Recurrent Neural Networks:

- RNNs are a type of artificial neural network that excel at processing sequential data, such as time series, natural language, speech, and more.
- Unlike feedforward neural networks, RNNs have feedback connections, allowing them to maintain an internal memory of past inputs and use it to process current inputs.
- The main advantage of RNNs is their ability to capture temporal dependencies and handle variable-length sequences.
- All 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.





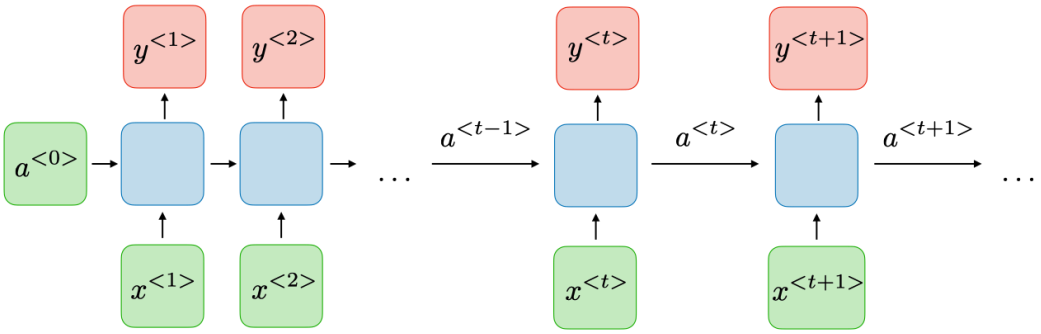
Recurrent Neural Networks Architecture:

The fundamental feature of a Recurrent Neural Network (RNN) is that the network contains at least one feed-back connection, so the activations can flow round in a loop. That enables the networks to do temporal processing and learn sequences, e.g., perform sequence recognition/reproduction or temporal association/prediction.

Recurrent neural network architectures can have many different forms. One common type consists of a standard Multi-Layer Perceptron (MLP) plus added loops. These can exploit the powerful non-linear mapping capabilities of the MLP, and have some form of memory. Others have more uniform structures, potentially with every neuron connected to all the others, and may also have stochastic activation functions.

- RNNs are very powerful, because they combine two properties:
  - – Distributed hidden state that allows them to store a lot of information about the past efficiently.
  - – Non-linear dynamics that allows them to update their hidden state in complicated ways.
- With enough neurons and time, RNNs can compute anything that can be computed by your computer.

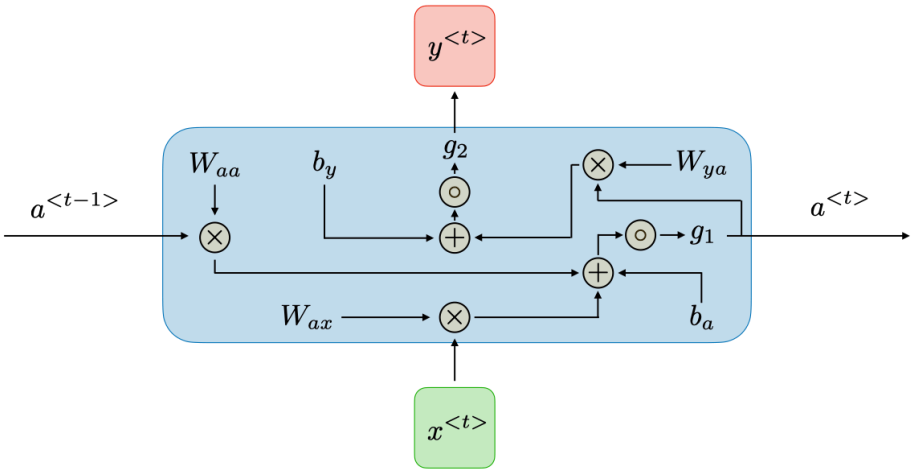
RNNs are a type of neural network that has hidden states and allows past outputs to be used as inputs.



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

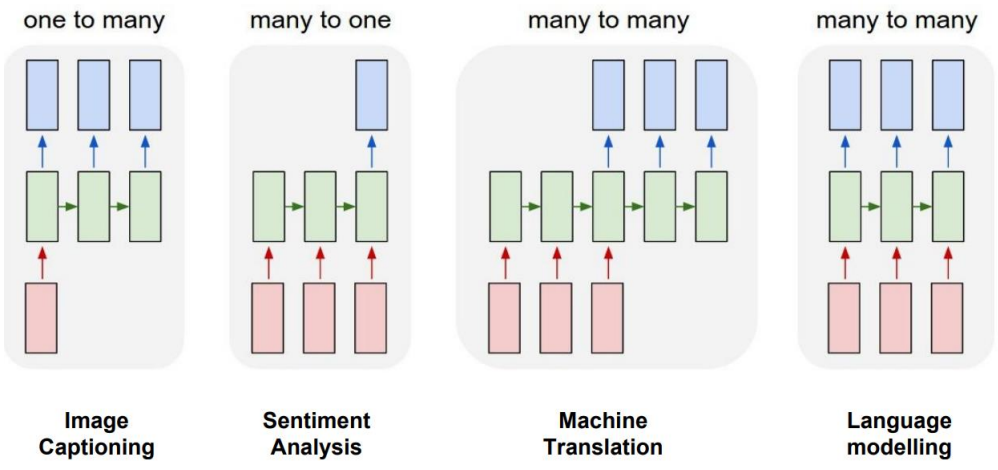
$$a^{<t>} = g_1(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a) \quad \text{and} \quad y^{<t>} = g_2(W_{ya}a^{<t>} + b_y)$$

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



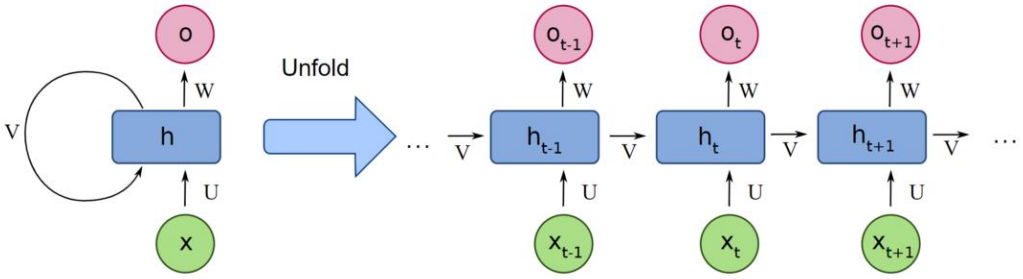
RNN architecture can vary depending on the problem you're trying to solve. From those with a single input and output to those with many (with variations between).

- **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.

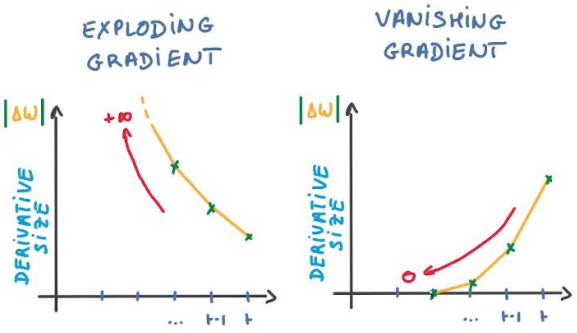


- The input layer x receives and processes the neural network's input before passing it on to the middle layer.
- Multiple hidden layers can be found in the middle layer h, each with its own activation functions, weights, and biases. You can utilize a recurrent neural network if the various parameters 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.

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.

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.



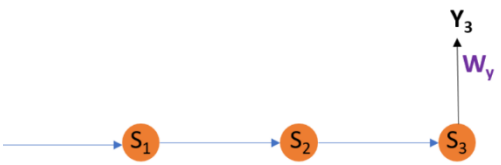
- **Exploding Gradients:** Exploding gradients occur when the algorithm gives the weights an absurdly high priority for no apparent reason. Fortunately, truncating or squashing the gradients is a simple solution to this problem.
- **Vanishing Gradients:** Vanishing gradients occur when the gradient values 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.

Back Propagation through time – RNN

To perform back propagation, we must adjust the weights associated with inputs, the memory units and the outputs.

Adjusting Wy :

For better understanding, let us consider the following representation:



Formula:

$$\frac{\partial E_3}{\partial W_y} = \frac{\partial E_3}{\partial Y_3} \cdot \frac{\partial Y_3}{\partial W_Y}$$

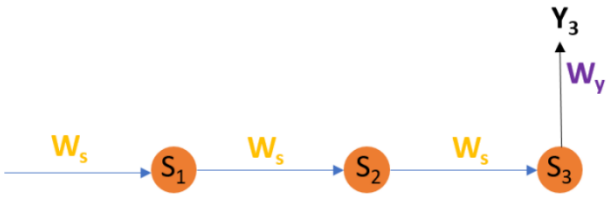
Explanation:

E_3 is a function of Y_3. Hence, we differentiate E_3 w.r.t Y_3.

Y_3 is a function of W_Y. Hence, we differentiate Y_3 w.r.t W_Y.

Adjusting Ws

For better understanding, let us consider the following representation:



Formula:

$$\frac{\partial E_3}{\partial W_S} = (\frac{\partial E_3}{\partial Y_3}.\frac{\partial Y_3}{\partial S_3}.\frac{\partial S_3}{\partial W_S})+$$

$$(\frac{\partial E_3}{\partial Y_3}.\frac{\partial Y_3}{\partial S_3}.\frac{\partial S_3}{\partial S_2}.\frac{\partial S_2}{\partial W_S})+$$

$$(\frac{\partial E_3}{\partial Y_3}.\frac{\partial Y_3}{\partial S_3}.\frac{\partial S_3}{\partial S_2}.\frac{\partial S_2}{\partial S_1}.\frac{\partial S_1}{\partial W_S})$$

Explanation:

E3 is a function of Y3. Hence, we differentiate E3 w.r.t Y3.
Y3 is a function of S3. Hence, we differentiate Y3 w.r.t S3.
S3 is a function of WS. Hence, we differentiate S3 w.r.t WS.
But we cannot stop with this; we also have to take into consideration, the previous time steps. So, we differentiate (partially) the Error function with respect to memory units S2 as well as S1 taking into consideration the weight matrix WS.
We must keep in mind that a memory unit, say St is a function of its previous memory unit St-1.

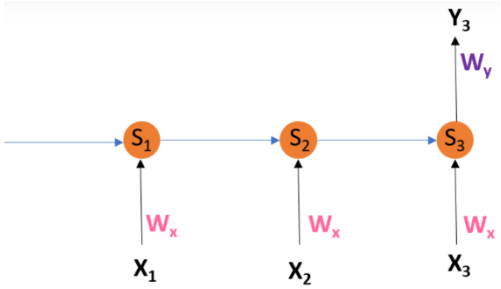
Hence, we differentiate S3 with S2 and S2 with S1.

Generally, we can express this formula as:

$$\boxed{\frac{\partial E_N}{\partial W_S} = \sum_{i=1}^{N} \frac{\partial E_N}{\partial Y_N}.\frac{\partial Y_N}{\partial S_i}.\frac{\partial S_i}{\partial W_S}}$$

Adjusting WX:

For better understanding, let us consider the following representation:



$$\frac{\partial E_3}{\partial W_X} = (\frac{\partial E_3}{\partial Y_3}.\frac{\partial Y_3}{\partial S_3}.\frac{\partial S_3}{\partial W_X})+$$

$$(\frac{\partial E_3}{\partial Y_3}.\frac{\partial Y_3}{\partial S_3}.\frac{\partial S_3}{\partial S_2}.\frac{\partial S_2}{\partial W_X})+$$

$$(\frac{\partial E_3}{\partial Y_3}.\frac{\partial Y_3}{\partial S_3}.\frac{\partial S_3}{\partial S_2}.\frac{\partial S_2}{\partial S_1}.\frac{\partial S_1}{\partial W_X})$$

Explanation:

E_3 is a function of Y_3. Hence, we differentiate E_3 w.r.t Y_3.
Y_3 is a function of S_3. Hence, we differentiate Y_3 w.r.t S_3.
S_3 is a function of W_X. Hence, we differentiate S_3 w.r.t W_X.

Again, we cannot stop with this; we also have to take into consideration, the previous time steps. So, we differentiate (partially) the Error function with respect to memory units S2 as well as S1 taking into consideration the weight matrix W_X.

Generally, we can express this formula as:

$$\boxed{\frac{\partial E_N}{\partial W_S} = \sum_{i=1}^{N} \frac{\partial E_N}{\partial Y_N}.\frac{\partial Y_N}{\partial S_i}.\frac{\partial S_i}{\partial W_X}}$$

Advantages of Recurrent Neural Network

1. An RNN remembers each piece of information through time. It is useful in time series prediction only because of the feature to remember previous inputs as well. This is called Long Short-Term Memory.
2. Recurrent neural networks are even used with convolutional layers to extend the effective pixel neighborhood.

Disadvantages of Recurrent Neural Network

1. Gradient vanishing and exploding problems.
2. Training an RNN is a very difficult task.
3. It cannot process very long sequences if using tanh or relu as an activation function.

Applications of Recurrent Neural Network

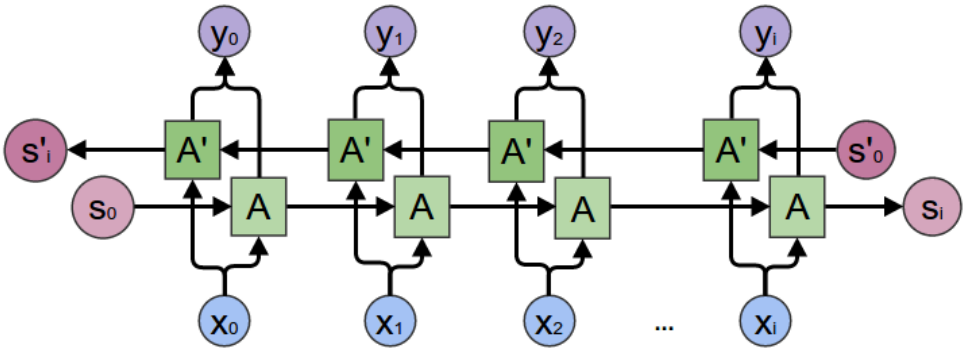
1. Language Modelling and Generating Text
2. Speech Recognition
3. Machine Translation
4. Image Recognition, Face detection
5. Time series Forecasting

Bidirectional RNN:

- A bi-directional recurrent neural network (Bi-RNN) is a type of recurrent neural network (RNN) that processes input data in both forward and backward directions.
- The goal of a Bi-RNN is to capture the contextual dependencies in the input data by processing it in both directions, which can be useful in a variety of natural language processing (NLP) tasks.
- For example, in natural language processing tasks, a uni-directional RNN may not accurately predict the next word in a sentence if the previous words provide important context for the current word.

  Consider an example where we could use the recurrent network to predict the masked word in a sentence.
  - Apple is my favorite _____.
  - Apple is my favourite _____, and I work there.
  - Apple is my favorite _____, and I am going to buy one.

  In the first sentence, the answer could be fruit, company, or phone. But in the second and third sentences, it cannot be a fruit.



- This allows the network to consider information from the past and future when making predictions rather than just relying on the input data at the current time step.
- This can be useful for tasks such as language processing, where understanding the context of a word or phrase can be important for making accurate predictions.
- In general, bidirectional RNNs can help improve the performance of a model on a variety of sequence-based tasks.
- During the forward pass of the RNN, the forward RNN processes the input sequence in the usual way by taking the input at each time step and using it to update the hidden state. The updated hidden state is then used to predict the output at that time step.
- During the backward pass, the backward RNN processes the input sequence in reverse order and makes

predictions for the output sequence. These predictions are then compared to the target output sequence in reverse order, and the error is backpropagated through the network to update the weights of the backward RNN.

- One common way to combine the outputs of the forward and reverse RNNs is to concatenate them, but other methods, such as element-wise addition or multiplication can also be used. The choice of combination method can depend on the specific task and the desired properties of the final output.
- Once both passes are complete, the weights of the forward and backward RNNs are updated based on the errors computed during the forward and backward passes, respectively. This process is repeated for multiple iterations until the model converges and the predictions of the bidirectional RNN are accurate.
- This allows the bidirectional RNN to consider information from past and future time steps when making predictions, which can significantly improve the model's accuracy.

Building RNN for Text Classification

Simple Bidirectional RNN for Sentiment Analysis

Import:

Importing and loading the dataset required libraries to perform the sentiment analysis tasks.

```
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np

# Load the IMDB Reviews dataset
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.imdb.load_data(num_words=10000)
```

Padding:

Padding is a common technique used in natural language processing (NLP) to ensure that all input sequences have the same length. This is often necessary because many NLP models, such as neural networks, require fixed-length input sequences.

```
# Pad the sequences to have equal length
max_len = 500
x_train = tf.keras.preprocessing.sequence.pad_sequences(x_train, maxlen=max_len)
x_test = tf.keras.preprocessing.sequence.pad_sequences(x_test, maxlen=max_len)
```

Build the Model:

```
# Set the input and output dimensions
input_dim = 10000
output_dim = 1
# Create the input layer
inputs = tf.keras.Input(shape=(None,), dtype="int32")

# Create the model
x = tf.keras.layers.Embedding(input_dim, 128)(inputs)
x = tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(64, return_sequences=True))(x)
x = tf.keras.layers.Bidirectional(tf.keras.layers.LSTM(64))(x)
outputs = tf.keras.layers.Dense(output_dim, activation="sigmoid")(x)
model = tf.keras.Model(inputs, outputs)

# Compile the model
model.compile("adam", "binary_crossentropy", metrics=["accuracy"])
```

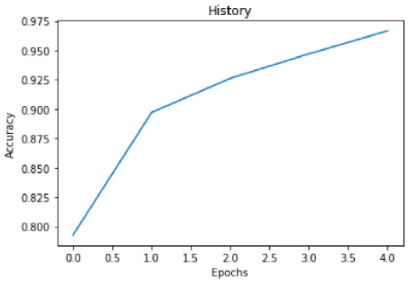
Train

```
# Train the model
batch_size = 32
epochs = 5
history = model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs, validation_data=(x_test,
```

```
y_test))
```

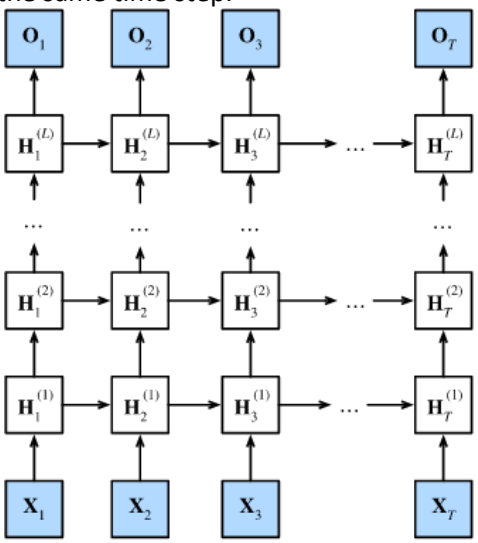
Evaluate the accuracy

```
# Plot the accuracy
fig = plt.plot(history.history['accuracy'])
title = plt.title("History")
xlabel = plt.xlabel("Epochs")
ylabel = plt.ylabel("Accuracy")
```



Deep recurrent neural networks:

- Despite having just one hidden layer between the input at any time step and the corresponding output, there is a sense in which these networks are deep.
- Inputs from the first-time step can influence the outputs at the final time step T (often 100s or 1000s of steps later). These inputs pass through T applications of the recurrent layer before reaching the final output.
- The standard method for building this sort of deep RNN is strikingly simple: we stack the RNNs on top of each other. Given a sequence of length T, the first RNN produces a sequence of outputs, also of length T.
- These, in turn, constitute the inputs to the next RNN layer. In this short section, we illustrate this design pattern and present a simple example for how to code up such stacked RNNs. Below, in Fig, we illustrate a deep RNN with L hidden layers.
- Each hidden state operates on a sequential input and produces a sequential output. Moreover, any RNN cell (white box in Fig) at each time step depends on both the same layer's value at the previous time step and the previous layer's value at the same time step.



Formally, suppose that we have a minibatch input $\mathbf{X}_t \in \mathbb{R}^{n \times d}$ (number of examples: n, number of inputs in each example: d) at time step t. At the same time step, let the hidden state of the l^{th} hidden layer ($l = 1, \ldots, L$) be $\mathbf{H}_t^{(l)} \in \mathbb{R}^{n \times h}$ (number of hidden units: h) and the output layer variable be $\mathbf{O}_t \in \mathbb{R}^{n \times q}$ (number of outputs: q). Setting $\mathbf{H}_t^{(0)} = \mathbf{X}_t$, the hidden state of the l^{th} hidden layer that uses the activation function ϕ_l is calculated as follows:

$$\mathbf{H}_t^{(l)} = \phi_l(\mathbf{H}_t^{(l-1)}\mathbf{W}_{xh}^{(l)} + \mathbf{H}_{t-1}^{(l)}\mathbf{W}_{hh}^{(l)} + \mathbf{b}_h^{(l)}),$$

where the weights $\mathbf{W}_{xh}^{(l)} \in \mathbb{R}^{h \times h}$ and $\mathbf{W}_{hh}^{(l)} \in \mathbb{R}^{h \times h}$, together with the bias $\mathbf{b}_h^{(l)} \in \mathbb{R}^{1 \times h}$, are the model parameters of the l^{th} hidden layer.

In the end, the calculation of the output layer is only based on the hidden state of the final L^{th} hidden layer:

$$\mathbf{O}_t = \mathbf{H}_t^{(L)}\mathbf{W}_{hq} + \mathbf{b}_q,$$

where the weight $\mathbf{W}_{hq} \in \mathbb{R}^{h\times q}$ and the bias $\mathbf{b}_q \in \mathbb{R}^{1\times q}$ are the model parameters of the output layer.

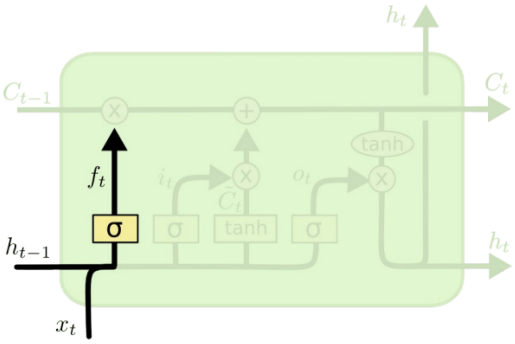
Long Short-Term Memory Networks:

To solve the problem of Vanishing and Exploding Gradients in a Deep Recurrent Neural Network, many variations were developed. One of the most famous of them is the Long Short Term Memory Network (LSTM). In concept, an LSTM recurrent unit tries to "remember" all the past knowledge that the network is seen so far and to "forget" irrelevant data. This is done by introducing different activation function layers called "gates" for different purposes. Each LSTM recurrent unit also maintains a vector called the Internal Cell State which conceptually describes the information that was chosen to be retained by the previous LSTM recurrent unit.

LSTM networks are the most used variation of Recurrent Neural Networks (RNNs). The critical component of the LSTM is the memory cell and the gates (including the forget gate but also the input gate), inner contents of the memory cell are modulated by the input gates and forget gates. If both segue, he is closed, the contents of the memory cell will remain unmodified between one time-step and the next gradients gating structure allows information to be retained across many time-steps, and consequently also allows group that to flow across many time-steps. This allows the LSTM model to overcome the vanishing gradient properly occurs with most Recurrent Neural Network models.

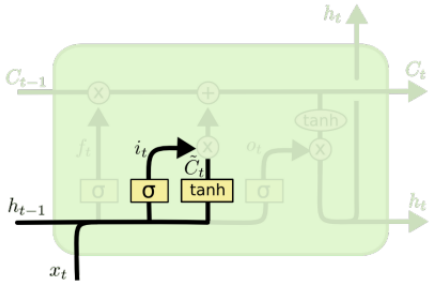
A Long Short Term Memory Network consists of four different gates for different purposes as described below: -

- **Forget Gate(f):** At forget gate the input is combined with the previous output to generate a fraction between 0 and 1, that determines how much of the previous state need to be preserved (or in other words, how much of the state should be forgotten). This output is then multiplied with the previous state. Note: An activation output of 1.0 means "remember everything" and activation output of 0.0 means "forget everything." From a different perspective, a better name for the forget gate might be the "remember gate"



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

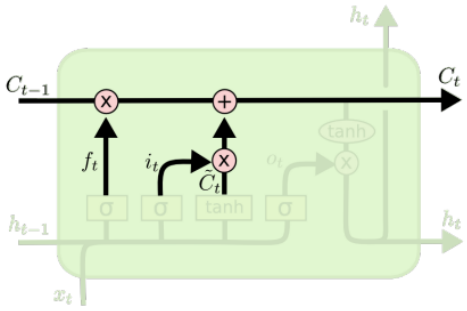
- **Input Gate(i):** Input gate operates on the same signals as the forget gate, but here the objective is to decide which new information is going to enter the state of LSTM. The output of the input gate (again a fraction between 0 and 1) is multiplied with the output of tan h block that produces the new values that must be added to previous state. This gated vector is then added to previous state to generate current state



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

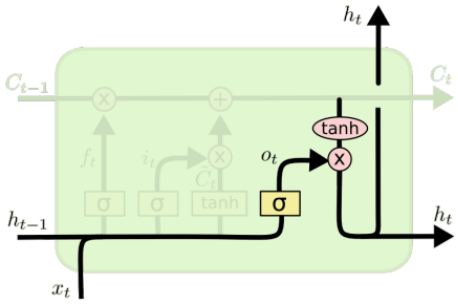
- **Memory Update (g):** It is often considered as a sub-part of the input gate and much literature on LSTM's does not even mention it and assume it is inside the Input gate. It is used to modulate the information that the Input gate will write onto the Internal State Cell by adding non-linearity to the information and

making the information Zero-mean. This is done to reduce the learning time as Zero-mean input has faster convergence. Although this gate's actions are less important than the others and are often treated as a finesse-providing concept, it is good practice to include this gate in the structure of the LSTM unit.



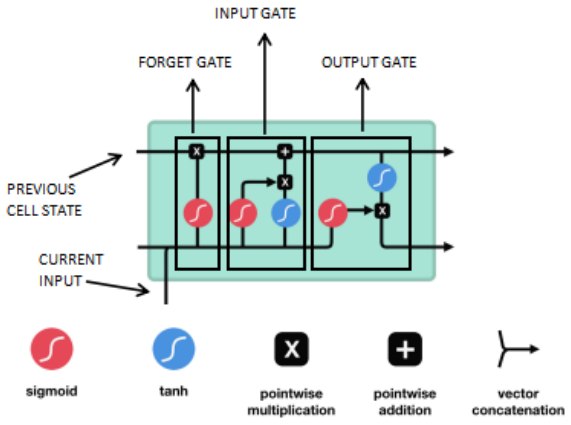
$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

- **Output Gate(o):** At output gate, the input and previous state are gated as before to generate another scaling fraction that is combined with the output of tanh block that brings the current state. This output is then given out. The output and state are fed back into the LSTM block.



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

Over all the LSTM diagram is



Advantages of LSTM over RNN:

- **Capturing Long-Term Dependencies:** LSTMs are designed to address the vanishing gradient problem in traditional RNNs, allowing them to capture and retain information over longer sequences. This makes them effective in modeling and understanding dependencies in time series, natural language processing, and other sequential data.
- **Memory Cell:** LSTMs have a memory cell that can store and retrieve information over extended time periods. This enables them to selectively remember or forget information based on the context, which is crucial for handling long-term dependencies.
- **Handling Variable-Length Sequences:** LSTMs can process variable-length sequences, making them suitable for tasks where the input lengths may vary, such as speech recognition, machine translation, and sentiment analysis.
- **Reduced Vanishing Gradient Problem:** The architecture of LSTMs includes gates that control the flow of information, such as the input gate, forget gate, and output gate. These gates regulate the

information flow and gradient propagation, mitigating the vanishing gradient problem and allowing better training of deep networks.

- **Versatility:** LSTMs can be used not only for prediction tasks but also for other tasks like sequence classification, anomaly detection, and generating new sequences. Their flexibility and ability to handle various types of sequential data make them widely applicable.

Disadvantages of LSTM over RNN:

- **Complexity and Training Time:** LSTMs are more complex than traditional RNNs, with additional gates and memory cells. This complexity results in increased training time and computational requirements, making them slower to train compared to simpler models.
- **Overfitting:** LSTMs, like other deep learning models, are prone to overfitting when trained on small datasets or when the model is excessively large. Regularization techniques and appropriate training strategies are necessary to mitigate this issue.
- **Difficulty in Interpretability:** The internal workings of LSTMs can be challenging to interpret and understand due to their complex structure. It can be difficult to gain insights into why the model makes certain predictions, which may be a concern in some applications where interpretability is important.
- **Need for Sufficient Data:** LSTM models typically require a sufficient amount of labeled training data to generalize well. Insufficient data can lead to poor performance or overfitting. Additionally, LSTMs may struggle with rare events or outliers if they are not well represented in the training data.
- **Parameter Tuning:** LSTMs have several hyperparameters, such as the number of layers, number of units per layer, learning rate, and regularization parameters. Finding the optimal values for these hyperparameters can be a challenging and time-consuming task.

	RNN	LSTM
Structure	Simple	More complex
Training	Can be difficult	Can be more difficult
Performance	Good for simple tasks	Good for complex tasks
Hidden state	Single	Multiple (memory cell)
Gates	None	Input, output, forget
Ability to retain long-term dependencies	Limited	Strong