UNIT-IV RECURRENT NEURAL NETWORKS

Recursive Neural Networks – Bidirectional RNNs – Deep Recurrent Networks – Applications: Image Generation, Image Compression, Natural Language Processing. Auto encoder -Complete Auto encoder, Regularized Autoencoder, LSTM

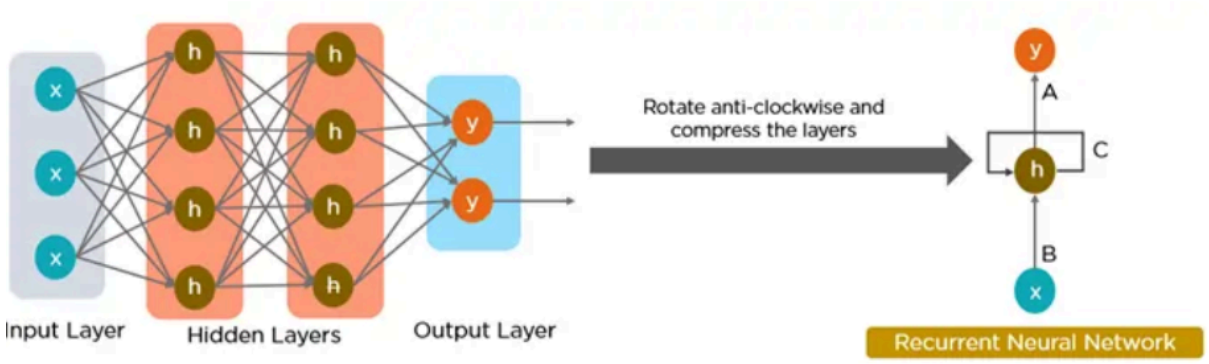
4.1Recurrent Neural Networks (RNN)

A recurrent neural network or RNN is a deep neural network trained on sequential or time series data to create a machine learning (ML) model that can make sequential predictions or conclusions based on sequential inputs.

Understanding Recurrent Neural Network (RNN)

Recurrent Neural Networks imitate the function of the human brain in the fields of Data science, Artificial intelligence, machine learning, and deep learning, allowing computer programs to recognize patterns and solve common issues.

RNNs are a type of neural network that can 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.



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. They employ the same settings for each input since they produce the same outcome by performing the same task on all inputs or hidden layers.

What Makes RNN Special?

Recurrent Neural Networks (RNNs) set themselves apart from other neural networks with their unique capabilities:

- **Internal Memory:** This is the key feature of RNNs. It allows them to remember past inputs and use that context when processing new information.

- **Sequential Data Processing:** Because of their memory, RNNs are exceptional at handling sequential data where the order of elements matters. This makes them ideal for speech recognition, machine translation, natural language processing (NLP), and text generation.
- **Contextual Understanding:** RNNs can analyse the current input about what they've "seen" before. This contextual understanding is crucial for tasks where meaning depends on prior information.
- **Dynamic Processing:** RNNs can continuously update their internal memory as they process new data, allowing them to adapt to changing patterns within a sequence.

RNN Architecture

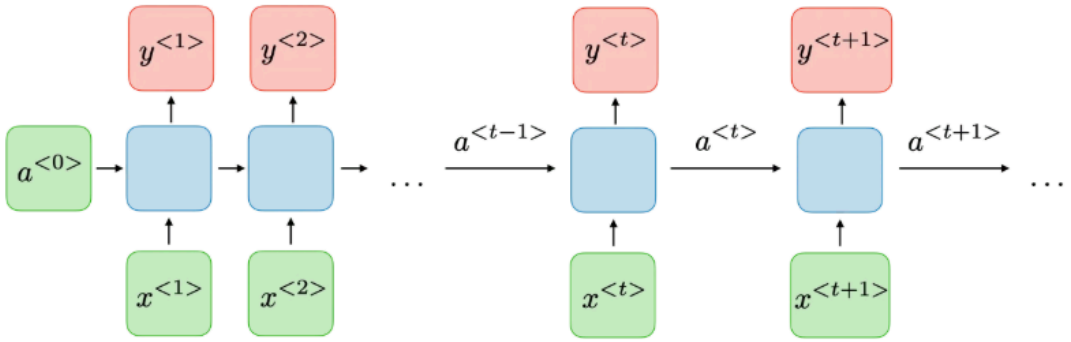
RNNs are a type of neural network that have hidden states and allow past outputs to be used as inputs. They usually follow a certain architecture. One example is the Deep RNN: by stacking multiple RNN layers on top of each other, deep RNNs create a more complex architecture. This allows them to capture intricate relationships within very long sequences of data. They are particularly useful for tasks where the order of elements spans long stretches.

Here's a breakdown of its key components:

- **Input Layer:** This layer receives the initial element of the sequence data. For example, in a sentence, it might receive the first word as a vector representation.
- **Hidden Layer:** The heart of the RNN, the hidden layer contains a set of interconnected neurons. Each neuron processes the current input along with the information from the previous hidden layer's state. This "state" captures the network's memory of past inputs, allowing it to understand the current element in context.
- **Activation Function:** This function introduces non-linearity into the network, enabling it to learn complex patterns. It transforms the combined input from the current input layer and the previous hidden layer state before passing it on.
- **Output Layer:** The output layer generates the network's prediction based on the processed information. In a language model, it might predict the next word in the sequence.
- **Recurrent Connection:** A key distinction of RNNs is the recurrent connection within the hidden layer. This connection allows the network to pass the hidden state information (the network's memory) to the next time step. It's like passing a baton in a relay race, carrying information about previous inputs forward

Architecture of a Traditional RNN

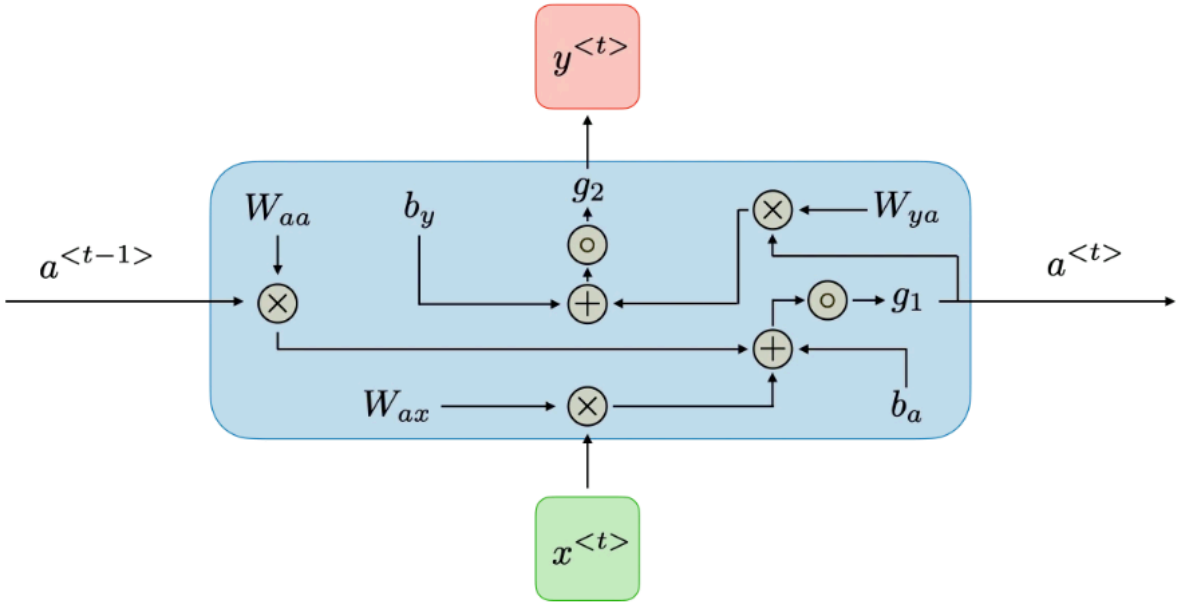
RNNs are a type of neural network with hidden states and allow past outputs to be used as inputs. They usually go like this:



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

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

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. It can range from those with a single input and output to those with many (with variations between).

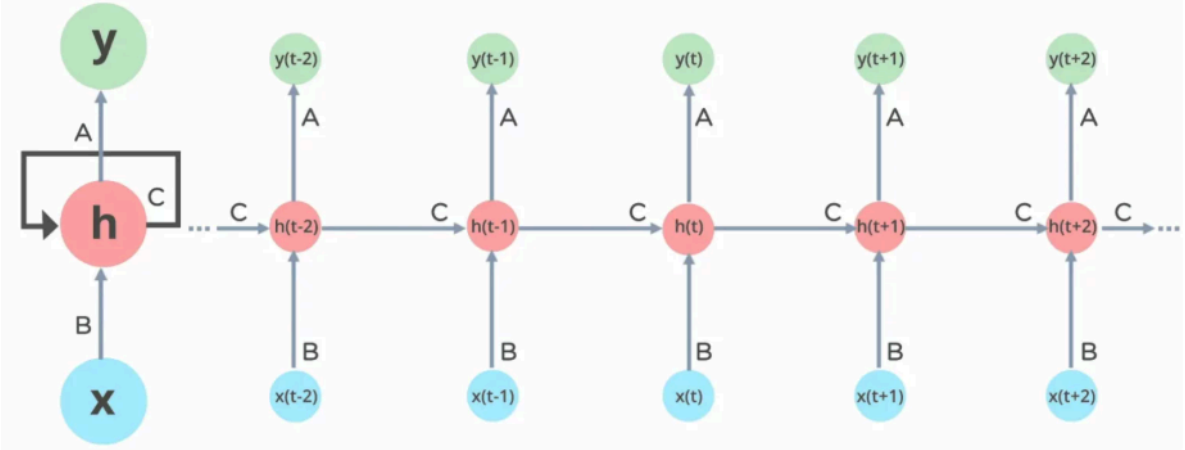
Below are some RNN architectures that can help you better understand this.

- **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 music production, for example.
- **Many-to-one:** A single output combines inputs from distinct time steps in this scenario. Sentiment analysis and emotion identification use such networks, in which a sequence of words determines the class label.

- **Many-to-Many:** For many-to-many, there are numerous options. Input and output are sequences of potentially different lengths. Machine translation systems, such as English to French or vice versa translation systems, use many-to-many networks.

How do Recurrent Neural Networks Work?

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

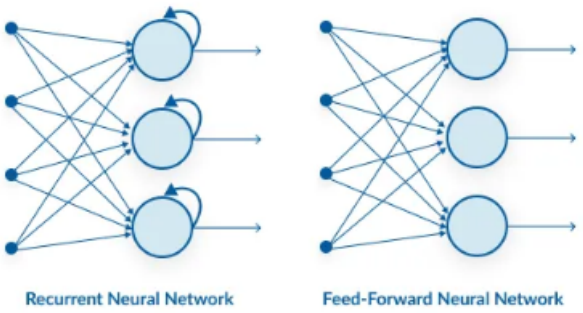


The input layer **x** receives and processes the neural network's input before passing it on to the middle layer.

In the middle layer h, multiple hidden layers can be found, each with its activation functions, weights, and biases. There's a single hidden layer that **repeats over time steps**. The same weights are used across all steps. The hidden state is updated recursively using the current input and the previous hidden state.

The recurrent neural network will standardize the different activation functions, weights, and biases, 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.

Recurrent Neural Network vs Feed-forward Neural Network



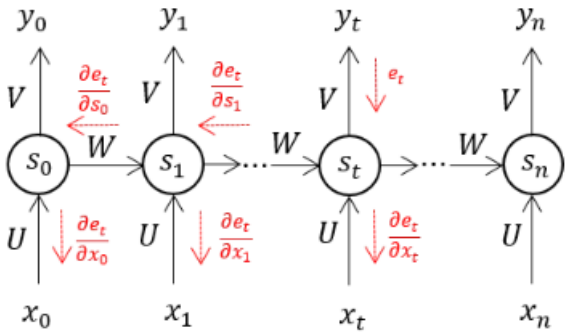
Feature	Feed-forward Neural Network (FNN)	Recurrent Neural Network (RNN)
Data Flow	Straight through (no loops)	Loops through time (with memory)

Feature	Feed-forward Neural Network (FNN)	Recurrent Neural Network (RNN)
Memory	No memory of past inputs	Maintains memory via hidden states
Input Assumption	Inputs are independent	Inputs are sequentially dependent
Architecture	Static layers, no time dimension	Unfolds across time steps
Training	Standard backpropagation	Backpropagation Through Time (BPTT)

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.

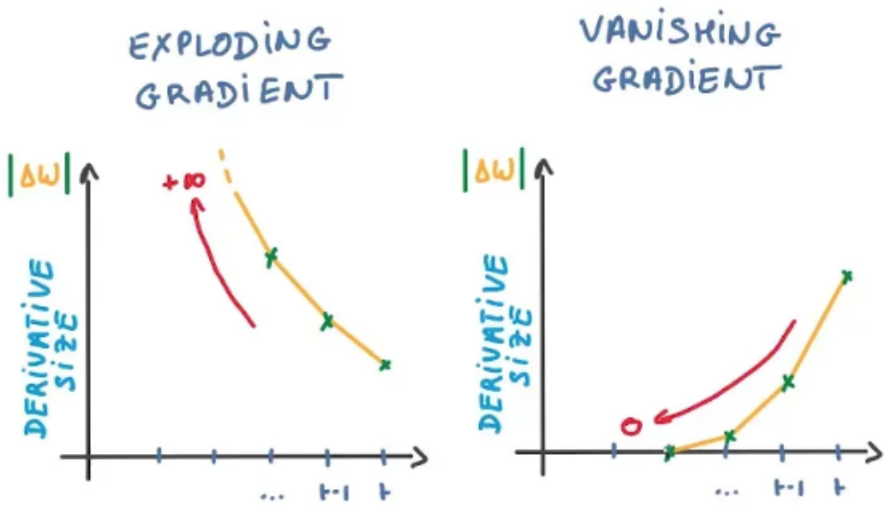
In a normal RNN, a single input is sent into the network at a time, and a single output is obtained. On the other hand, backpropagation uses both the current and prior inputs as input. This is referred to as a timestep, and one timestep will consist of multiple time series data points entering the RNN simultaneously.



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

Two Issues of Standard RNNs

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



About its inputs, a gradient is a partial derivative. If you're unsure 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, and 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 about 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.

What Are the Different Variations of RNN?

Researchers have introduced new, advanced RNN architectures to overcome issues like vanishing and exploding gradient descent that hinder learning in long sequences.

- **Long Short-Term Memory (LSTM):** A popular choice for complex tasks. LSTM networks introduce gates, i.e., input gate, output gate, and forget gate, that control the flow of information within the network, allowing them to learn long-term dependencies more effectively than vanilla RNNs.
- **Gated Recurrent Unit (GRU):** Similar to LSTMs, GRUs use gates to manage information flow. However, they have a simpler architecture, making them faster to train while maintaining good performance. This makes them a good balance between complexity and efficiency.
- **Bidirectional RNN:** This variation processes data in both forward and backward directions. This allows it to capture context from both sides of a sequence, which is useful for tasks like sentiment analysis where understanding the entire sentence is crucial.
- **Deep RNN:** Stacking multiple RNN layers on top of each other, deep RNNs create a more complex architecture. This allows them to capture intricate relationships within very long

sequences of data. They are particularly useful for tasks where the order of elements spans long stretches.

RNN Applications

Recurrent neural networks (RNNs) shine in tasks involving sequential data, where order and context are crucial. Let's explore some real-world use cases. Using RNN models and sequence datasets, you may tackle a variety of problems, including :

- **Speech Recognition:** RNNs power virtual assistants like Siri and Alexa, allowing them to understand spoken language and respond accordingly.
- **Machine Translation:** RNNs translate languages more accurately, like Google Translate by analysing sentence structure and context.
- **Text Generation:** RNNs are behind chatbots that can hold conversations and even creative writing tools that generate different text formats.
- **Time Series Forecasting:** RNNs analyze financial data to predict stock prices or weather patterns based on historical trends.
- **Music Generation:** RNNs can generate music by learning patterns from existing pieces and generating new melodies or accompaniments.
- **Video Captioning:** RNNs analyze video content and automatically generate captions, making video browsing more accessible.
- **Anomaly Detection:** RNNs can learn normal patterns in data streams (e.g., network traffic) and detect anomalies that might indicate fraud or system failures.
- **Sentiment Analysis:** RNNs can analyze sentiment in social media posts, reviews, or surveys by understanding the context and flow of text.
- **Stock Market Recommendation:** RNNs can analyze market trends and news to suggest potential investment opportunities.
- **Sequence study of the genome and DNA:** RNNs can analyze sequential data in genomes and DNA to identify patterns and predict gene function or disease risk.

Advantages and Disadvantages of RNNs

Advantages of RNNs	Disadvantages of RNNs
Handle sequential data effectively, including text, speech, and time series.	Prone to vanishing and exploding gradient problems, hindering learning.
Process inputs of any length, unlike feedforward neural networks.	Training can be challenging, especially for long sequences.
Share weights across time steps, enhancing training efficiency.	Computationally slower than other neural network architectures.

Basic Python Implementation (RNN with Keras)

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.

```
import numpy as np

import tensorflow as tf

from tensorflow import keras

from tensorflow.keras import layers

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
```

4.2 Bidirectional Recurrent Neural Networks

What is 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 various natural language processing (NLP) tasks.

In a Bi-RNN, the input data is passed through two separate RNNs: one processes the data in the forward direction, while the other processes it in the reverse direction. The outputs of these two RNNs are then combined in some way to produce the final output.

One common way to combine the outputs of the forward and reverse RNNs is to concatenate them. Still, 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.

Need for Bi-directional RNNs

- A uni-directional recurrent neural network (RNN) processes input sequences in a single direction, either from left to right or right to left.
- This means the network can only use information from earlier time steps when making predictions at later time steps.
- This can be limiting, as the network may not capture important contextual information relevant to the output prediction.

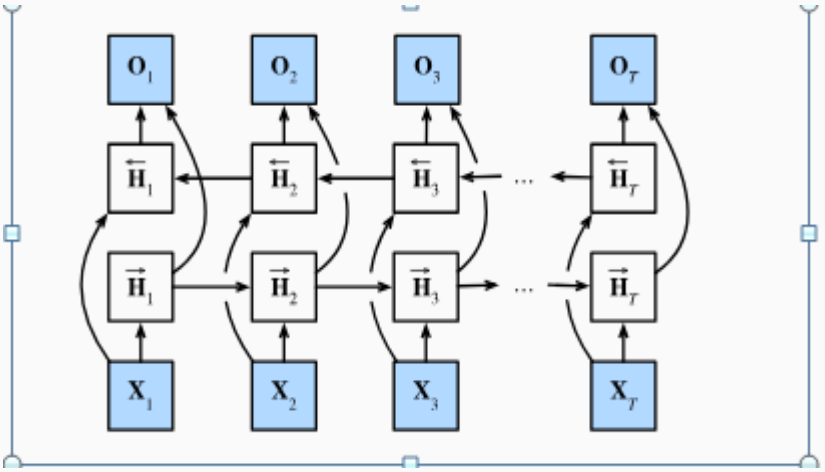
So far, our working example of a sequence learning task has been language modeling, where we aim to predict the next token given all previous tokens in a sequence. In this scenario, we wish only to condition upon the leftward context, and thus the unidirectional chaining of a standard RNN seems appropriate. However, there are many other sequence learning tasks contexts where it is perfectly fine to condition the prediction at every time step on both the leftward and the rightward context. Consider, for example, part of speech detection. Why shouldn't we take the context in both directions into account when assessing the part of speech associated with a given word?

Another common task—often useful as a pretraining exercise prior to fine-tuning a model on an actual task of interest—is to mask out random tokens in a text document and then to train a sequence model to predict the values of the missing tokens. Note that depending on what comes after the blank, the likely value of the missing token changes dramatically:

- I am ___.
- I am ___ hungry.
- I am ___ hungry, and I can eat half a pig.

In the first sentence “happy” seems to be a likely candidate. The words “not” and “very” seem plausible in the second sentence, but “not” seems incompatible with the third sentences.

Fortunately, a simple technique transforms any unidirectional RNN into a bidirectional RNN (Schuster and Paliwal, 1997). We simply implement two unidirectional RNN layers chained together in opposite directions and acting on the same input (Fig. 10.4.1). For the first RNN layer, the first input is and the last input is , but for the second RNN layer, the first input is and the last input is . To produce the output of this bidirectional RNN layer, we simply concatenate together the corresponding outputs of the two underlying unidirectional RNN layers.



Architecture of a bidirectional RNN.

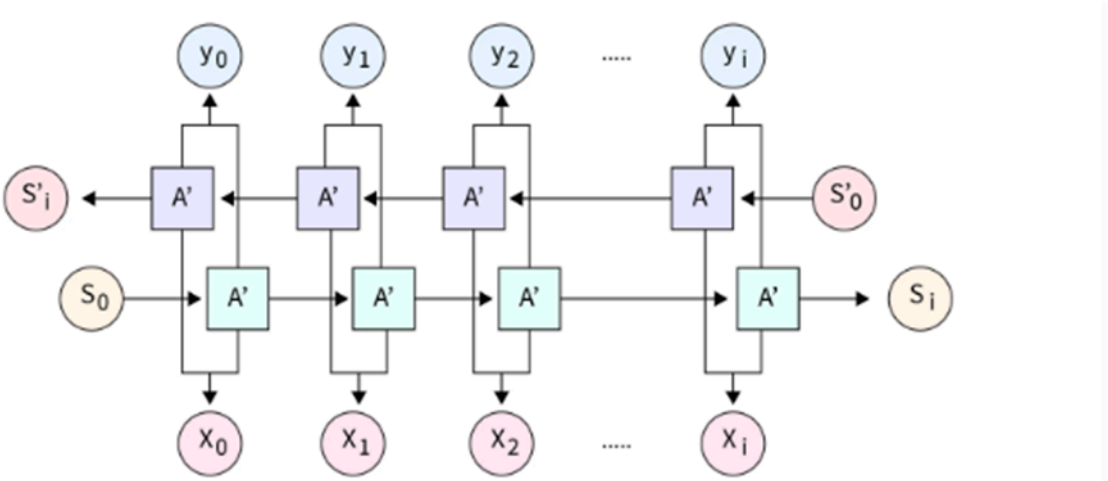
Formally for any time step , we consider a minibatch input (number of examples ; number of inputs in each example) and let the hidden layer activation function be . In the bidirectional architecture, the forward and backward hidden states for this time step are and , respectively, where is the number of hidden units.

Bi-directional RNNs

- A bidirectional recurrent neural network (RNN) is a type of recurrent neural network (RNN) that processes input sequences in both forward and backward directions.
- This allows the RNN to capture information from the input sequence that may be relevant to the output prediction. Still, the same could be lost in a traditional RNN that only processes the input sequence in one direction.
- 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 a model's performance on various sequence-based tasks.

This means that the network has **two separate RNNs**:

1. One that processes the input sequence from left to right
2. Another one that processes the input sequence from right to left.



These two RNNs are typically called forward and backward RNNs, respectively.

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.

Backpropagation through time (BPTT) is a widely used algorithm for training recurrent neural networks (RNNs). It is a variant of the backpropagation algorithm specifically

designed to handle the temporal nature of RNNs, where the output at each time step depends on the inputs and outputs at previous time steps.

In the case of a bidirectional RNN, BPTT involves two separate Backpropagation passes: one for the forward RNN and one for the backward RNN. During the forward pass, the forward RNN processes the input sequence in the usual way and makes predictions for the output sequence. These predictions are then compared to the target output sequence, and the error is backpropagated through the network to update the weights of the forward RNN.

The backward RNN processes the input sequence in reverse order during the backward pass and predicts 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.

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.

Merge Modes in Bidirectional RNN

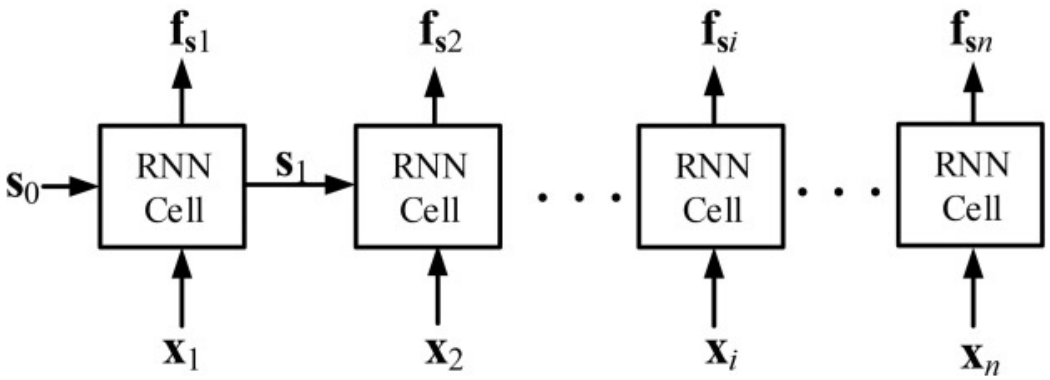
In a bidirectional recurrent neural network (RNN), two separate RNNs process the input data in opposite directions (forward and backward). The output of these two RNNs is then combined, or "merged," in some way to produce the final output of the model.

There are several ways in which the outputs of the forward and backward RNNs can be merged, depending on the specific needs of the model and the task it is being used for. Some common merge modes include:

1. **Concatenation:** In this mode, the outputs of the forward and backward RNNs are concatenated together, resulting in a single output tensor that is twice as long as the original input.
2. **Sum:** In this mode, the outputs of the forward and backward RNNs are added together element-wise, resulting in a single output tensor that has the same shape as the original input.
3. **Average:** In this mode, the outputs of the forward and backward RNNs are averaged element-wise, resulting in a single output tensor that has the same shape as the original input.

4. **Maximum:** In this mode, the maximum value of the forward and backward outputs is taken at each time step, resulting in a single output tensor with the same shape as the original input.

Which merge mode to use will depend on the specific needs of the model and the task it is being used for. Concatenation is generally a good default choice and works well in many cases, but other merge modes may be more appropriate for certain tasks.



4.3 Deep Recurrent Neural Networks (DRNNs):

Deep Recurrent Neural Networks (DRNNs) are a type of recurrent neural network (RNN) that uses multiple hidden layers to process sequential data, enabling them to learn more complex patterns and representations than single-layer RNNs. By stacking multiple layers, deep RNNs can capture different levels of abstraction in the sequential data, making them powerful for tasks like natural language processing, time series analysis, and speech recognition.

1. What are Recurrent Neural Networks?

- **RNNs are a type of neural network designed to handle sequential data, where the order of elements matters.**
- **Unlike traditional feedforward networks, RNNs have connections that loop back, allowing them to retain information about previous inputs (memory).**
- **This "memory" enables RNNs to capture temporal dependencies and contextual information in sequential data, making them suitable for tasks like language modeling, time series prediction, and speech recognition.**

2. What are Deep Recurrent Neural Networks?

- **Deep RNNs extend the basic RNN architecture by adding multiple hidden layers.**

- **Each layer in a deep RNN can learn increasingly complex representations of the sequential data, allowing the network to capture intricate patterns and relationships.**
- **This increased depth can significantly improve performance compared to single-layer RNNs, especially on complex tasks.**

3. Key Concepts in Deep RNNs:

Recurrent Connections:
The core feature of RNNs, allowing information to be passed from one step to the next within the network.
Hidden State:
A memory component within the RNN that stores information about previous inputs, enabling the network to maintain context.
Deep Architecture:
The stacking of multiple hidden layers in a deep RNN allows for hierarchical feature extraction and more powerful representation learning.

4. Applications of Deep RNNs:

- **Natural Language Processing (NLP): Language translation, text generation, sentiment analysis, and more.**
- **Speech Recognition: Converting spoken language into text.**
- **Time Series Analysis: Predicting future values in a sequence, such as stock prices or weather patterns.**
- **Image Captioning: Generating textual descriptions of images.**
- **Music Generation: Creating melodies and harmonies.**

4.4 Autoencoders:

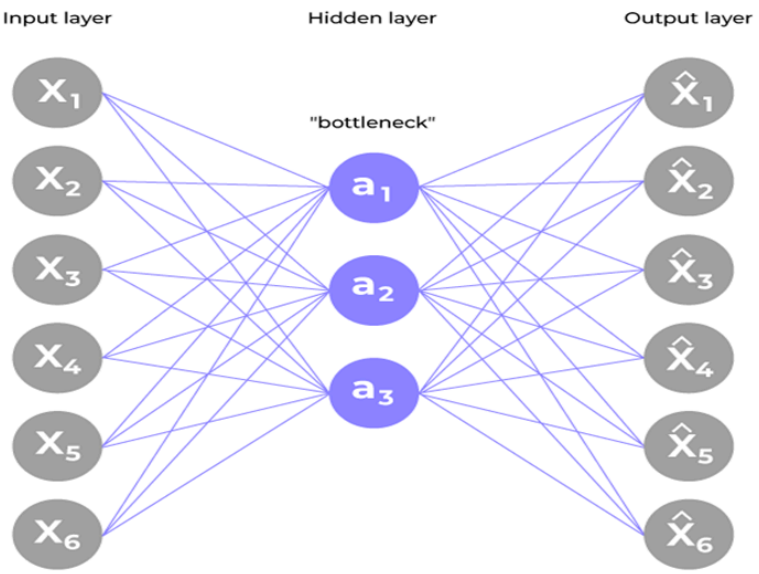
Autoencoders are a special type of **neural networks** that learn to compress data into a compact form and then reconstruct it to closely match the original input. They consist of an:

- **Encoder** that captures important features by reducing dimensionality.
- **Decoder** that rebuilds the data from this compressed representation.

The model trains by minimizing reconstruction error using loss functions like **Mean Squared Error** or **Binary Cross-Entropy**. These are applied in tasks such as noise removal, error detection and feature extraction where capturing efficient data representations is important.

Architecture of Autoencoder:

An autoencoder's architecture consists of three main components that work together to compress and then reconstruct data which are as follows:



1. Encoder

It compress the input data into a smaller, more manageable form by reducing its dimensionality while preserving important information. It has three layers which are:

- **Input Layer**: This is where the original data enters the network. It can be images, text features or any other structured data.
- **Hidden Layers**: These layers perform a series of transformations on the input data. Each hidden layer applies weights and activation functions to capture important patterns, progressively reducing the data's size and complexity.
- **Output(Latent Space)**: The encoder outputs a compressed vector known as the **latent representation** or **encoding**. This vector captures the important features of the input data in a condensed form helps in filtering out noise and redundancies.

2. Bottleneck (Latent Space)

It is the smallest layer of the network which represents the most compressed version of the input data. It serves as the information bottleneck which force the network to prioritize the most significant features. This compact representation helps the model learn the underlying structure and key patterns of the input helps in enabling better generalization and efficient data encoding.

3. Decoder

It is responsible for taking the compressed representation from the latent space and reconstructing it back into the original data form.

- **Hidden Layers**: These layers progressively expand the latent vector back into a higher-dimensional space. Through successive transformations decoder attempts to restore the original data shape and details
- **Output Layer**: The final layer produces the reconstructed output which aims to closely resemble the original input. The quality of reconstruction depends on how well the encoder-decoder pair can minimize the difference between the input and output during training.

Loss Function in Autoencoder Training

During training an autoencoder's goal is to minimize the reconstruction loss which measures how different the reconstructed output is from the original input. The choice of loss function depends on the type of data being processed:

- **Mean Squared Error (MSE)**: This is commonly used for continuous data. It measures the average squared differences between the input and the reconstructed data.
- **Binary Cross-Entropy**: Used for binary data (0 or 1 values). It calculates the difference in probability between the original and reconstructed output.

During training the network updates its weights using backpropagation to minimize this reconstruction loss. By doing this it learns to extract and retain the most important features of the input data which are encoded in the latent space.

Efficient Representations in Autoencoders

Constraining an autoencoder helps it learn meaningful and compact features from the input data which leads to more efficient representations. After training only the encoder part is used to encode similar data for future tasks. Various techniques are used to achieve this are as follows:

- **Keep Small Hidden Layers**: Limiting the size of each hidden layer forces the network to focus on the most important features. Smaller layers reduce redundancy and allows efficient encoding.
- **Regularization**: Techniques like L1 or L2 regularization add penalty terms to the loss function. This prevents overfitting by removing excessively large weights which helps in ensuring the model to learns general and useful representations.
- **Denoising:** In denoising autoencoders **random noise** is added to the input during training. It learns to remove this noise during reconstruction which helps it focus on core, noise-free features and helps in improving robustness.
- **Tuning the Activation Functions**: Adjusting activation functions can promote sparsity by activating only a few neurons at a time. This sparsity reduces model complexity and forces the network to capture only the most relevant features.

Types of Autoencoders:

Lets see different types of Autoencoders which are designed for specific tasks with unique features:

1. Denoising Autoencoder

Denoising Autoencoder is trained to handle corrupted or noisy inputs, it learns to remove noise and helps in reconstructing clean data. It prevent the network from simply memorizing the input and encourages learning the core features.

2. Sparse Autoencoder

Sparse Autoencoder contains more hidden units than input features but only allows a few neurons to be active simultaneously. This sparsity is controlled by zeroing some hidden units, adjusting activation functions or adding a sparsity penalty to the loss function.

3. Variational Autoencoder

Variational autoencoder (VAE) makes assumptions about the probability distribution of the data and tries to learn a better approximation of it. It uses stochastic gradient descent to optimize and learn the distribution of latent variables. They used for generating new data such as creating realistic images or text.

It assumes that the data is generated by a Directed Graphical Model and tries to learn an approximation to $q\phi(z|x)$ $q\phi(z|x)$ to the conditional property $q\theta(z|x)$ $q\theta(z|x)$ where ϕ ϕ and θ θ are the parameters of the encoder and the decoder respectively.

4. Convolutional Autoencoder

Convolutional autoencoder uses convolutional neural networks (CNNs) which are designed for processing images. The encoder extracts features using convolutional layers and the decoder reconstructs the image through **deconvolution** also called as upsampling.

Limitations of Autoencoders:

Autoencoders are useful but also have some limitations:

1. **Memorizing Instead of Learning Patterns**: It can sometimes memorize the training data rather than learning meaningful patterns which reduces their ability to generalize to new data.

2. **Reconstructed Data Might Not Be Perfect**: Output may be blurry or distorted with noisy inputs or if the model architecture lacks sufficient complexity to capture all details.

3. **Requires a Large Dataset and Good Parameter Tuning**: It require large amounts of data and careful parameter tuning (latent dimension size, learning rate, etc) to perform well.

Insufficient data or poor tuning can result in weak feature representations.

4.5 Regularized Autoencoder:

Regularization in autoencoders is crucial for preventing overfitting and improving generalization to unseen data. It works by adding a penalty term to the autoencoder's loss function, encouraging the model to learn more robust and meaningful representations rather than simply memorizing the training data.

Why Regularize Autoencoders?

Autoencoders, especially deep ones, can be prone to overfitting, particularly when the number of parameters is large compared to the training data size. Overfitting leads to poor performance on new, unseen data. Regularization techniques help mitigate this by:

- Preventing memorization:

By adding a penalty, regularization discourages the autoencoder from simply mapping inputs to outputs without learning any underlying structure or meaningful representation.

- Improving generalization:

Regularization encourages the autoencoder to learn a more robust and generalizable representation that can handle variations and noise in the input data.

- Controlling model complexity:

Regularization techniques can limit the complexity of the learned representation, preventing the model from becoming too specialized to the training data.

Common Regularization Techniques for Autoencoders:

- L1 and L2 Regularization:

These are the most common forms of regularization, adding penalties based on the sum of the absolute values (L1) or the sum of the squared values (L2) of the autoencoder's weights. L1 regularization can lead to sparse representations (where many weights are zero), while L2 regularization encourages smaller weights.

- Dropout:

Dropout randomly sets a fraction of the neurons to zero during training, preventing over-reliance on specific neurons and promoting more robust feature learning.

- Denoising Autoencoders:

These autoencoders are trained on corrupted (noisy) input data, forcing them to learn robust features that can reconstruct the original, uncorrupted data.

- Contractive Autoencoders:

These autoencoders add a penalty term to the loss function that encourages the learned representation to be less sensitive to small changes in the input, promoting robustness.

- Sparse Autoencoders:

These autoencoders encourage sparsity in the hidden layer activations, meaning that only a small fraction of neurons are active for any given input, leading to more efficient and meaningful representations.

- Geometric Regularization:

This type of regularization is used in cases where the data has inherent geometric structure. For example, Geometry Regularized Autoencoders (GRAE) enforce similarity between the learned representations and the underlying geometry of the data.

4.6 **LSTM:**

Long Short-Term Memory (LSTM) is an enhanced version of the Recurrent Neural Network (RNN) designed by Hochreiter and Schmidhuber. LSTMs can capture long-term dependencies in sequential data making them ideal for tasks like language translation, speech recognition and time series forecasting.

Unlike traditional RNNs which use a single hidden state passed through time LSTMs introduce a memory cell that holds information over extended periods addressing the challenge of learning long-term dependencies.

Problem with Long-Term Dependencies in RNN

Recurrent Neural Networks (RNNs) are designed to handle sequential data by maintaining a hidden state that captures information from previous time steps. However they often face challenges in learning long-term dependencies where information from distant time steps becomes crucial for making accurate predictions for current state. This problem is known as the vanishing gradient or exploding gradient problem.

- **Vanishing Gradient**: When training a model over time, the gradients which help the model learn can shrink as they pass through many steps. This makes it hard for the model to learn long-term patterns since earlier information becomes almost irrelevant.
- **Exploding Gradient**: Sometimes gradients can grow too large causing instability. This makes it difficult for the model to learn properly as the updates to the model become erratic and unpredictable.

Both of these issues make it challenging for standard RNNs to effectively capture long-term dependencies in sequential data.

LSTM Architecture

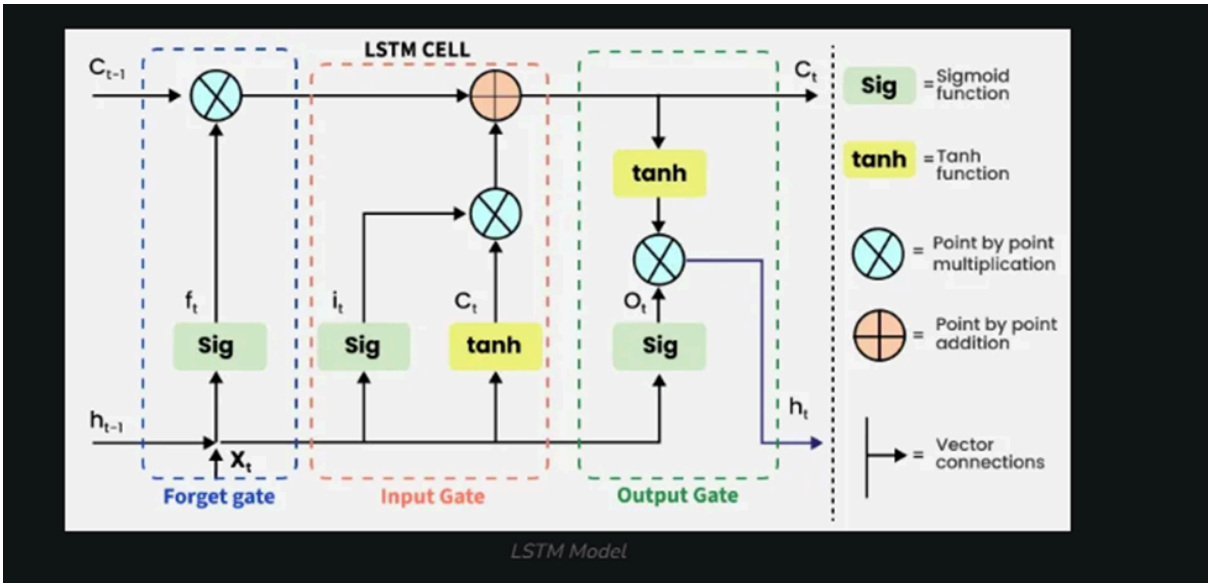
LSTM architectures involves the memory cell which is controlled by three gates:

1. **Input gate**: Controls what information is added to the memory cell.
2. **Forget gate**: Determines what information is removed from the memory cell.
3. **Output gate**: Controls what information is output from the memory cell.

This allows LSTM networks to selectively retain or discard information as it flows through the network which allows them to learn long-term dependencies. The network has a hidden state which is like its short-term memory. This memory is updated using the current input, the previous hidden state and the current state of the memory cell.

Working of LSTM

LSTM architecture has a chain structure that contains four neural networks and different memory blocks called cells.



Information is retained by the cells and the memory manipulations are done by the gates**.** There are three gates -

1. Forget Gate

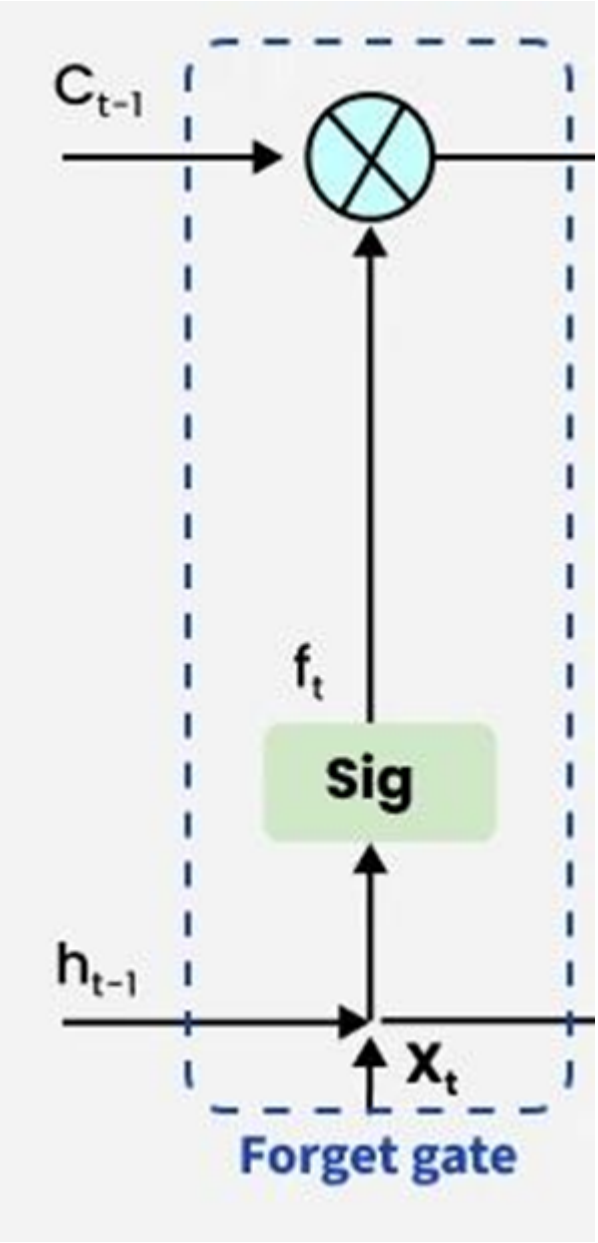
The information that is no longer useful in the cell state is removed with the forget gate. Two inputs xt*xt* (input at the particular time) and ht−1*ht*−1 (previous cell output) are fed to the gate and multiplied with weight matrices followed by the addition of bias. The resultant is passed through an activation function which gives a binary output. If for a particular cell state the output is 0, the piece of information is forgotten and for output 1, the information is retained for future use.

The equation for the forget gate is:

ft=σ(Wf⋅[ht−1,xt]+bf)*ft*=*σ*(*Wf*⋅[*ht*−1,*xt*]+*bf*)

Where:

- Wf*Wf*represents the weight matrix associated with the forget gate.
- [ht−1,xt][*ht*−1,*xt*] denotes the concatenation of the current input and the previous hidden state.
- bf*bf*is the bias with the forget gate.
- σσ is the sigmoid activation function



2. Input gate

The addition of useful information to the cell state is done by the input gate. First the information is regulated using the sigmoid function and filter the values to be remembered similar to the forget

gate using inputs ht−1*ht*−1and xt*xt*. Then, a vector is created using *tanh* function that gives an output from -1 to +1 which contains all the possible values from ht−1*ht*−1 and xt*xt*. At last the values of the vector and the regulated values are multiplied to obtain the useful information. The equation for the input gate is:

it=σ(Wi · [ht—1,xt]+bi)*it*=*σ*(*Wi* · [*ht*−1,*xt*]+*bi*)

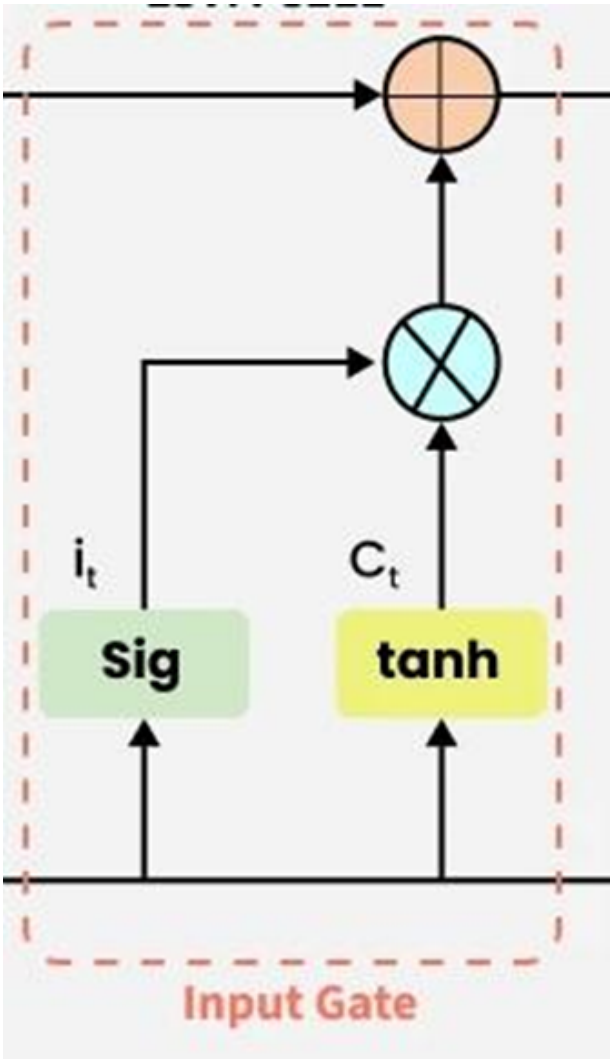
C^t=tanh(Wc·[ht–1,xt]+bc)*C*^*t*=tanh(*Wc* · [*ht*−1,*xt*]+*bc*)

We multiply the previous state by ft*ft* effectively filtering out the information we had decided to ignore earlier. Then we add it⊙Ct*it*⊙*Ct* which represents the new candidate values scaled by how much we decided to update each state value.

Ct=ft⊙Ct—1+it⊙C^t*Ct*=*ft*⊙*Ct*−1+*it*⊙*C*^*t*

where

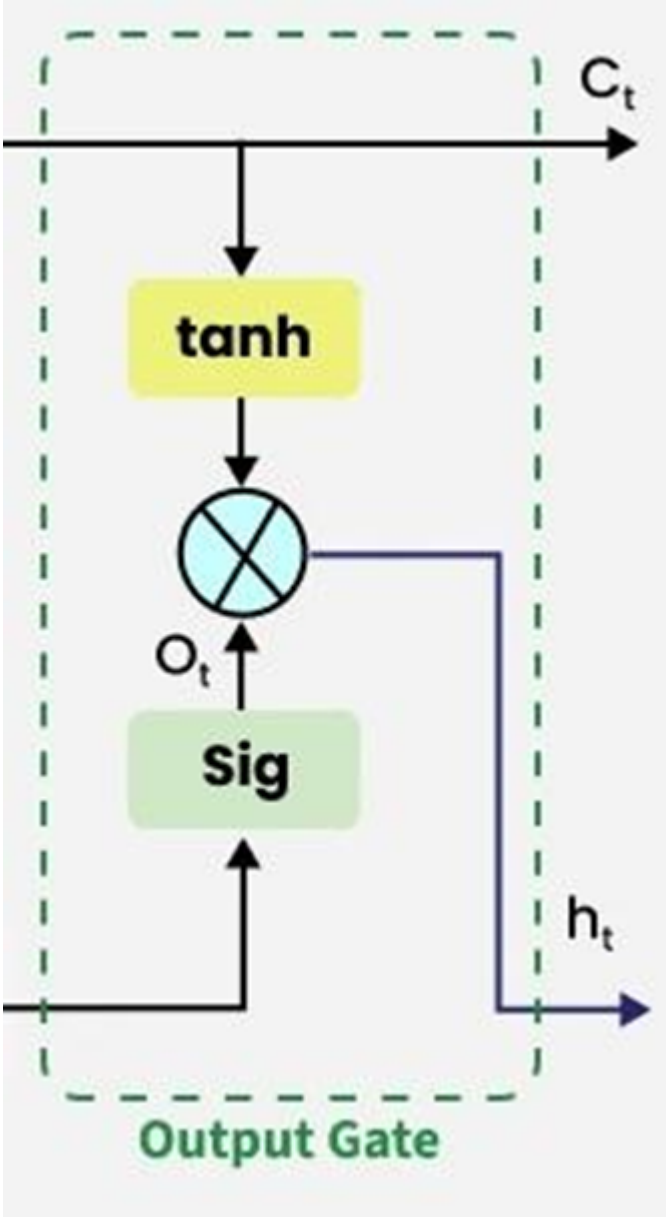
- ⊙⊙ denotes element-wise multiplication
- tanh is activation function



3. Output gate

The task of extracting useful information from the current cell state to be presented as output is done by the output gate. First, a vector is generated by applying tanh function on the cell. Then, the information is regulated using the sigmoid function and filter by the values to be remembered using inputsht−1*ht*−1and xt*xt*. At last the values of the vector and the regulated values are multiplied to be sent as an output and input to the next cell. The equation for the output gate is:

ot=σ(Wo · [ht—1,xt]+bo)*ot*=*σ*(*Wo* · [*ht*−1,*xt*]+*bo*)



Applications of LSTM

Some of the famous applications of LSTM includes:

- **Language Modeling**: Used in tasks like language modeling, machine translation and text summarization. These networks learn the dependencies between words in a sentence to generate coherent and grammatically correct sentences.
- **Speech Recognition**: Used in transcribing speech to text and recognizing spoken commands. By learning speech patterns they can match spoken words to corresponding text.
- **Time Series Forecasting**: Used for predicting stock prices, weather and energy consumption. They learn patterns in time series data to predict future events.
- **Anomaly Detection**: Used for detecting fraud or network intrusions. These networks can identify patterns in data that deviate drastically and flag them as potential anomalies.
- **Recommender Systems**: In recommendation tasks like suggesting movies, music and books. They learn user behavior patterns to provide personalized suggestions.
- **Video Analysis**: Applied in tasks such as object detection, activity recognition and action classification. When combined with Convolutional Neural Networks (CNNs) they help analyze video data and extract useful information.