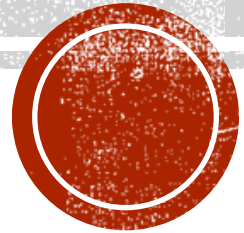
MODULE 4
RECURRENT NEURAL NETWORKS



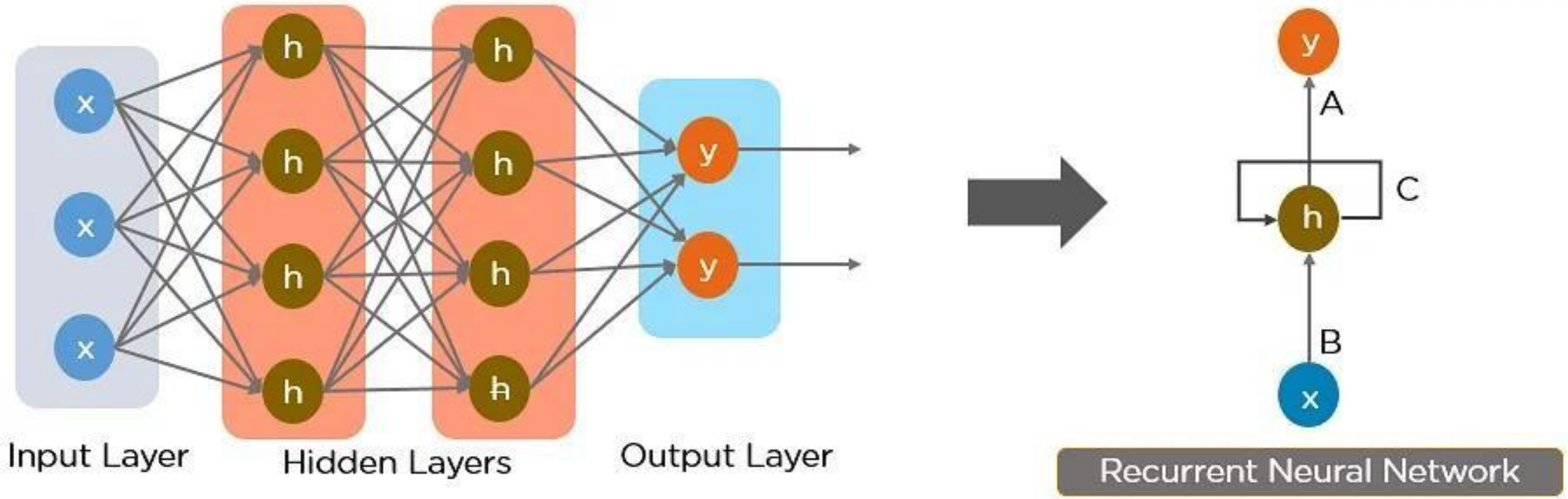
Dr Divya Meena Sundaram

Sr. Assistant Prof. Grade 2

SCOPE

VIT-AP University

Recurrent Neural Network(RNN)



Introduction to Recurrent Neural Network

- Recap:
  - In Deep Learning So far we complete Feed-Forward Neural Networks, Multilayer Perceptron, and Deep Neural Networks -(CNN)..
  - Feed forward - Pattern recognition and classification, non-linear regression, and function approximation. (Used for general Regression and Classification problems)
  - CNN - finding patterns in images to recognize objects, classes, and categories. (Used for object detection and image classification.)

Introduction to Recurrent Neural Network

- Recurrent Neural Network – Mainly focused on Language Modelling and Generating Text (Natural language processing (NLP))
- Example:
  - Google autocomplete predictions
  - Autocomplete is a feature for predicting the rest of a query a user is typing. Example (mail Composing in Gmail)

Introduction to Recurrent Neural Network

- Recurrent neural network is a type of neural network in which the output from the previous step is fed as input to the current step.
- RNN works on the principle of saving the output of a particular layer and feeding this back to the input in order to predict the output of the layer.

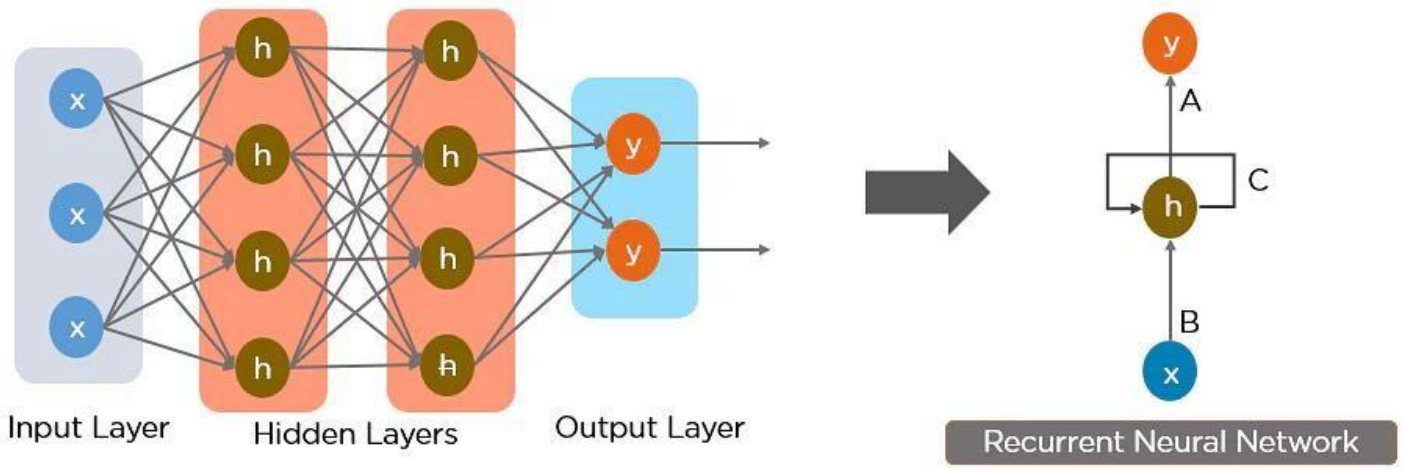
Why RNN?

- The basic challenge of classic feed-forward neural network is that it has no memory, that is, each training example given as input to the model is treated independent of each other.
- In order to work with sequential data with such models — you need to show them the entire sequence in one go as one training example.
- This is problematic because number of words in a sentence could vary and more importantly this is not how we tend to process a sentence in our head.

Recurrent Neural Network(RNN)

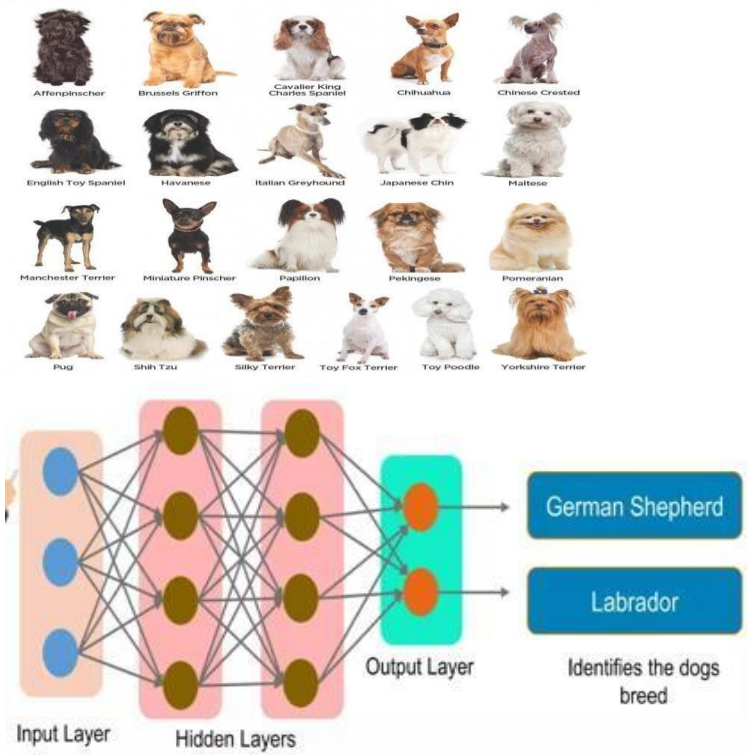
- RNN is so powerful since it doesn't take into consideration just the actual input but also the previous input which allows it to memorize what happens previously.
- Recurrent neural network is a type of neural network used to deal specifically with sequential data.
- State of the art algorithm for sequential data and are used by Apple's Siri and Google's voice search.
- It is the first algorithm that remembers its input, due to an internal memory, which makes it perfectly suited for machine learning problems that involve sequential data like time series, speech, text, financial data, audio, video, weather and much more.

- RNN is recurrent in nature as it performs the same function for every input of data while the output of the current input depends on the past one computation.
- For making a decision, it considers the current input and the output that it has learned from the previous input.



Let's Recap Feed Forward Network

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

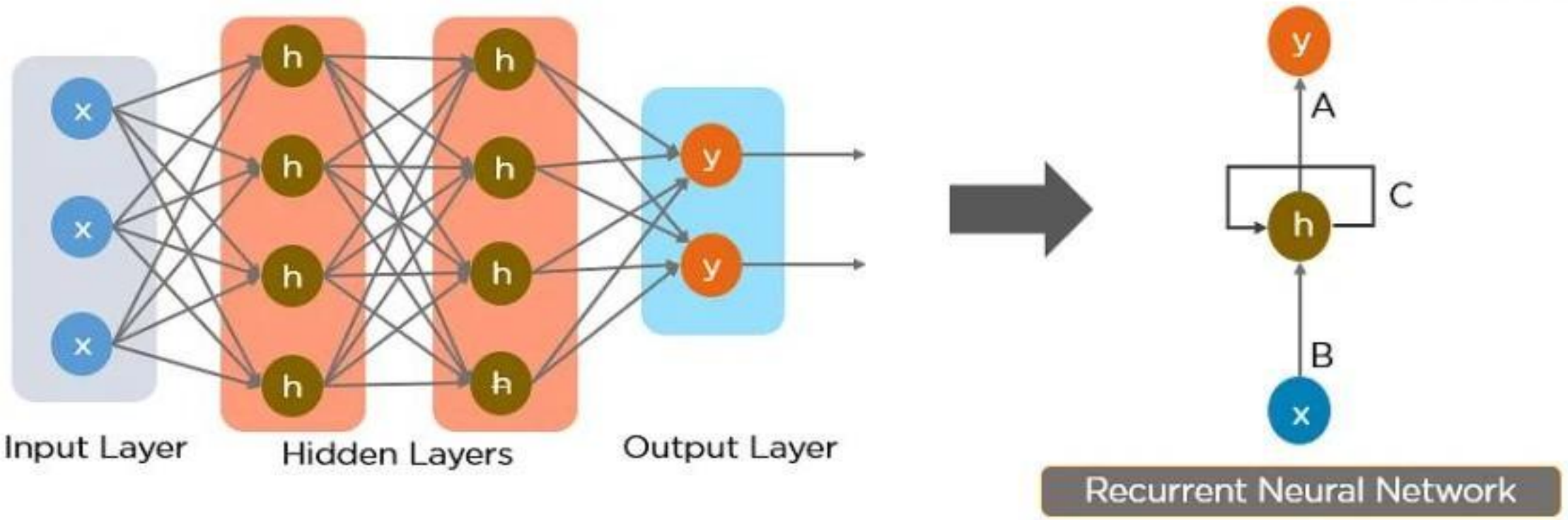


Here is an example of how neural networks can identify a dog's breed based on their features.

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

Now … Recurrent Neural Network (RNN)

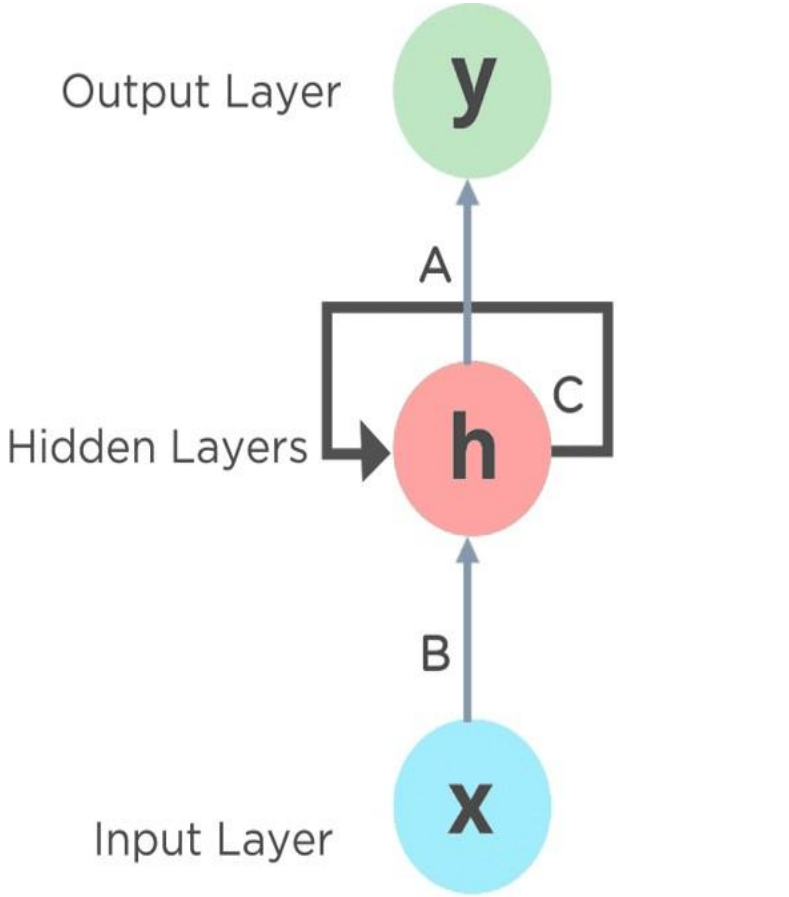
Feed forward



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

Now … Recurrent Neural Network (RNN)

Fully connected Recurrent Neural Network

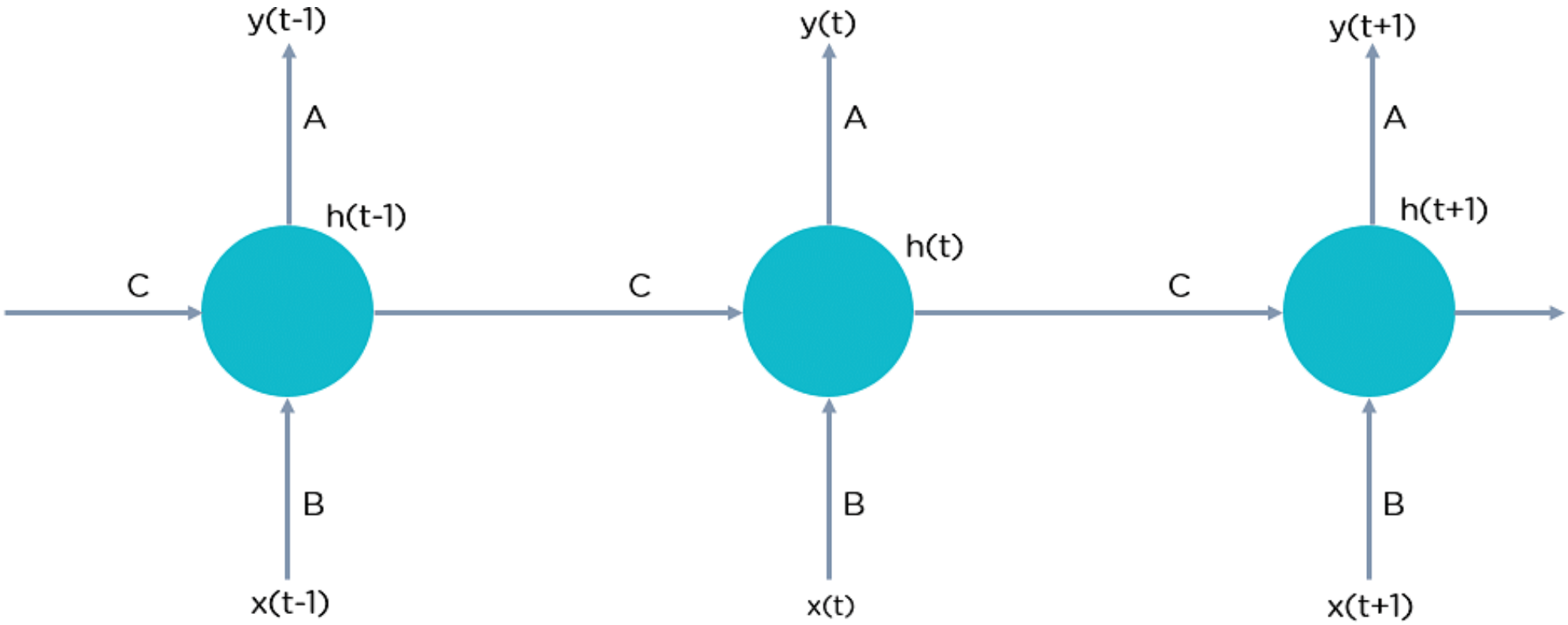


A, B and C are the parameters

- Here, "x" is the input layer, "h" is the hidden layer, and "y" is the output layer.
- A, B, and C are the network parameters used to improve the output of the model.
- At any given time t, the current input is a combination of input at x(t) and h(t-1).
- The output at any given time is fetched back to the network to improve on the output.

Now … Recurrent Neural Network (RNN)

Fully connected Recurrent Neural Network



$h(t) = f_c(h(t-1), x(t))$

h(t) = new state
f_c = function with parameter c
h(t-1) = old state
x(t) = input vector at time step t

Now … Recurrent Neural Network (RNN)

Now that you understand what a recurrent neural network.

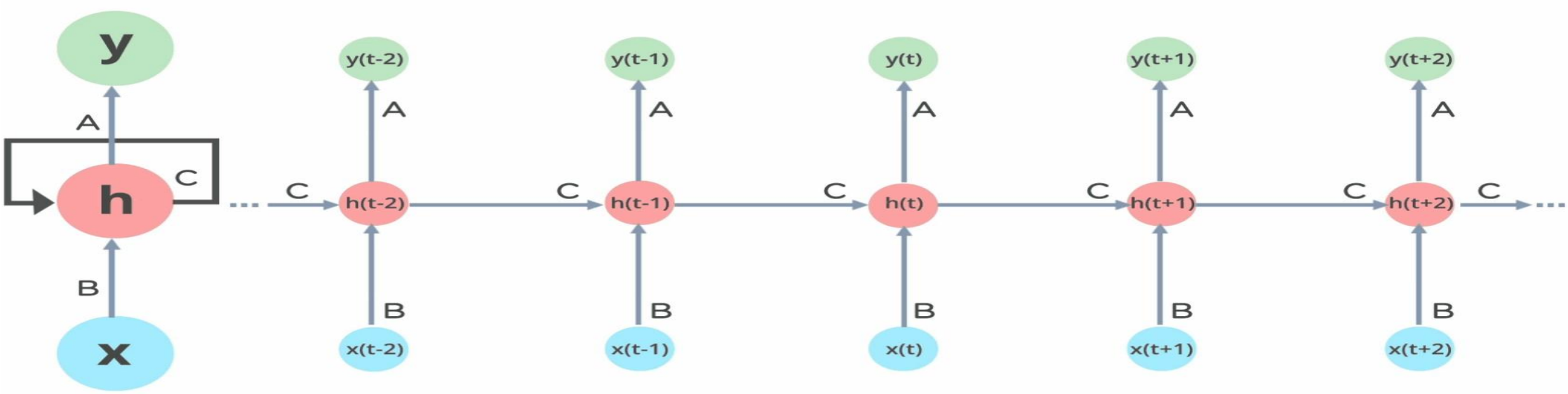
Why Recurrent Neural Networks?

- RNN were created because there were a few issues in the feed-forward neural network.
  - Cannot handle sequential data (Timeseries, stock price)
  - Considers only the current input
  - Cannot memorize previous inputs

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

How Does Recurrent Neural Networks Work?

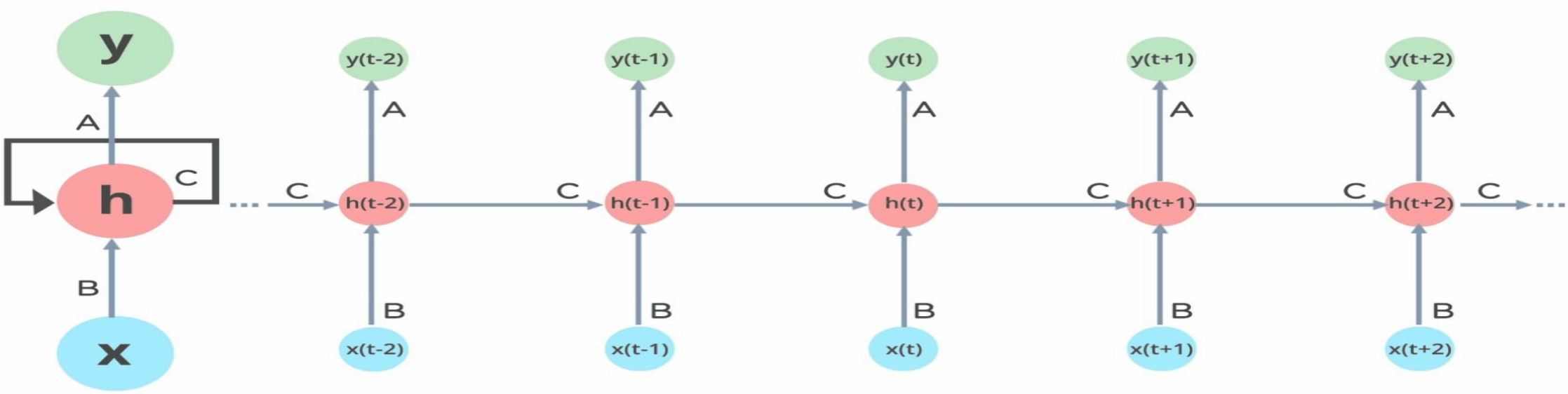
- In Recurrent Neural networks, the information cycles through a loop to the middle hidden layer.



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

How Does Recurrent Neural Networks Work?

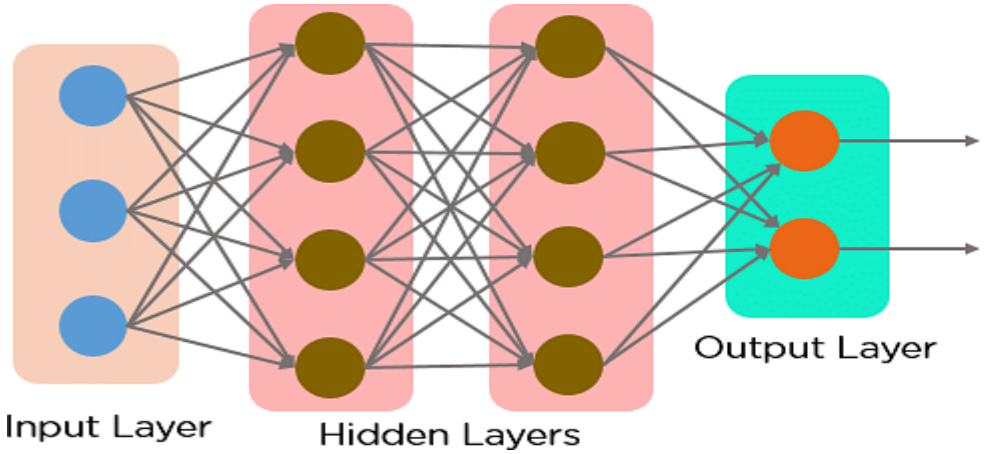
- In Recurrent Neural networks, the information cycles through a loop to the middle hidden layer.

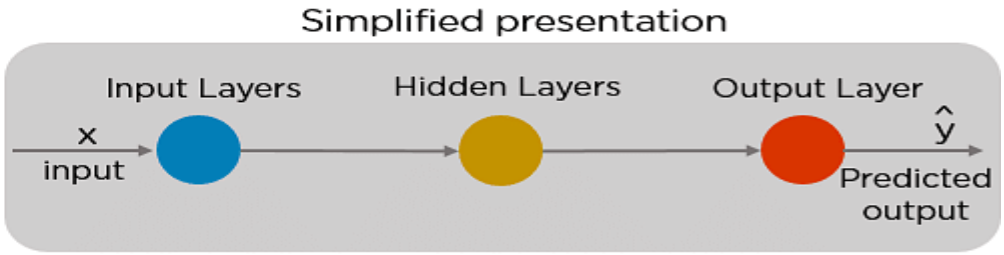


- The Recurrent Neural Network will standardize the different activation functions and weights and biases so that each hidden layer has the same parameters.
- Then, instead of creating multiple hidden layers, it will create one and loop over it as many times as required.

Feed-Forward Neural Networks vs RNN

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

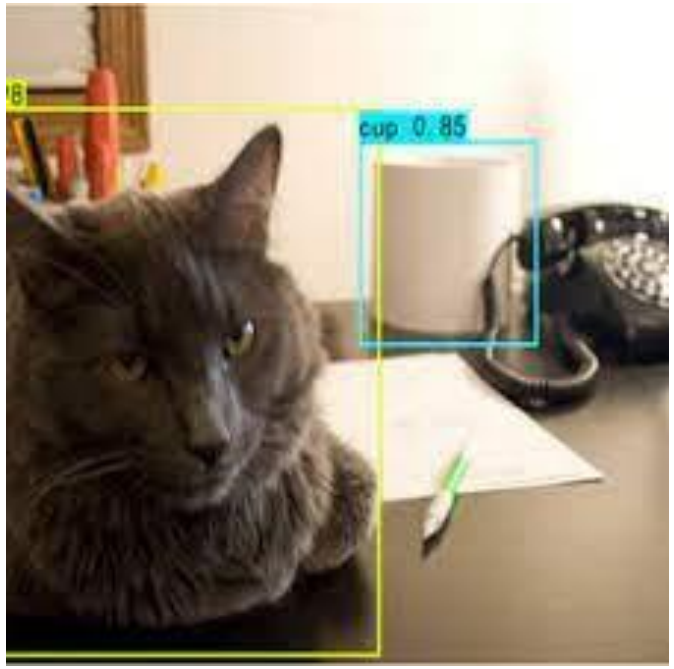




- In a feed-forward neural network, the decisions are based on the current input.
- It doesn't memorize the past data, and there's no future scope. Feed-forward neural networks are used in general regression and classification problems.

Applications of Recurrent Neural Networks

Image Captioning

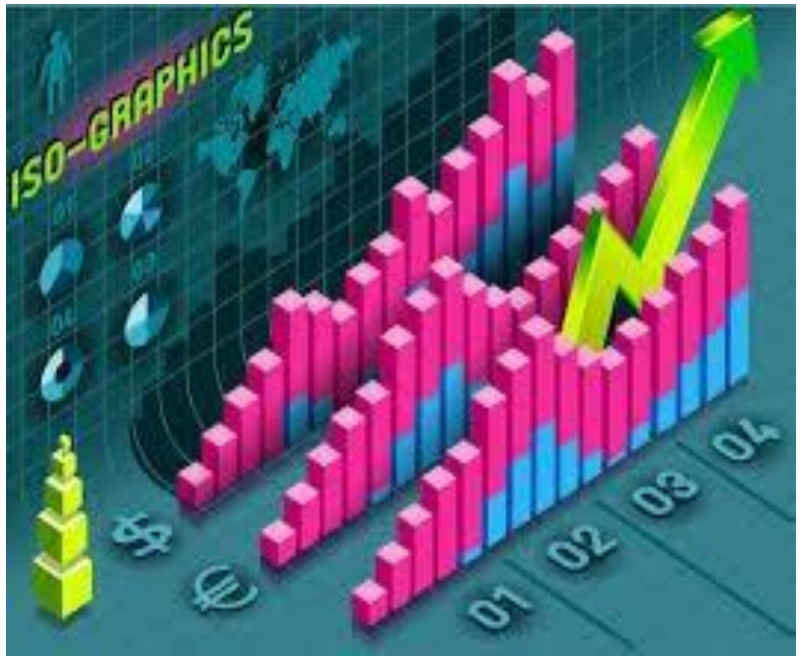


- RNNs are used to caption an image by analyzing the activities present.

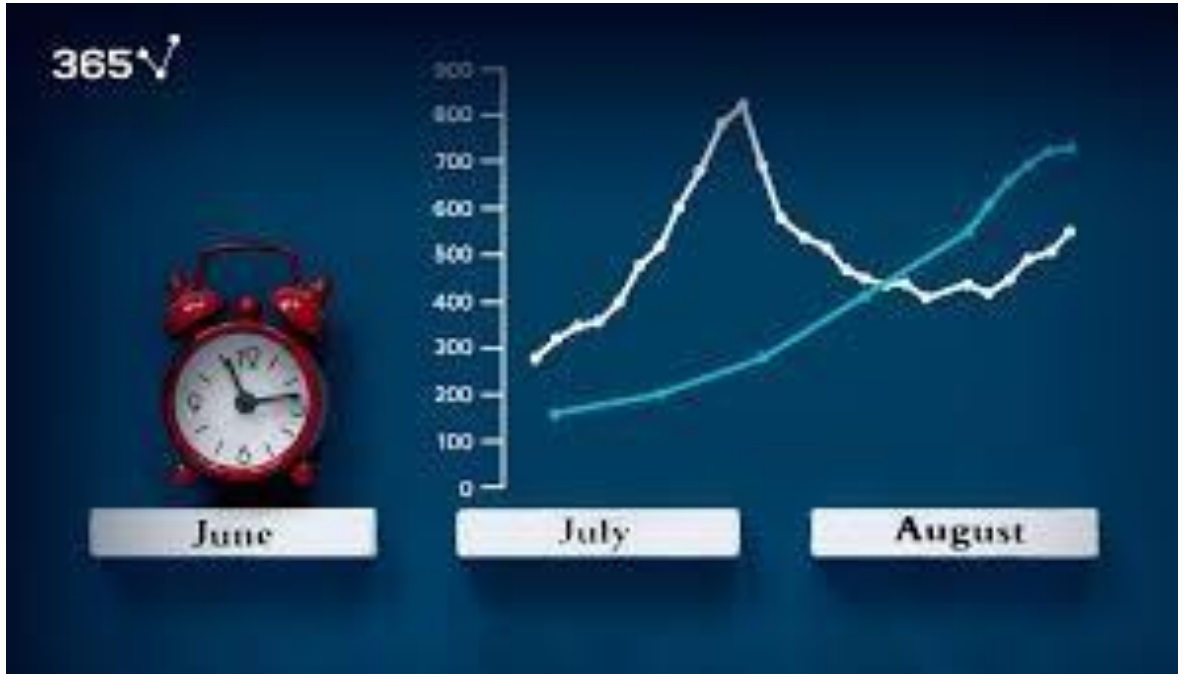
"A Dog catching a ball in mid air"

Applications of Recurrent Neural Networks

Time Series Prediction



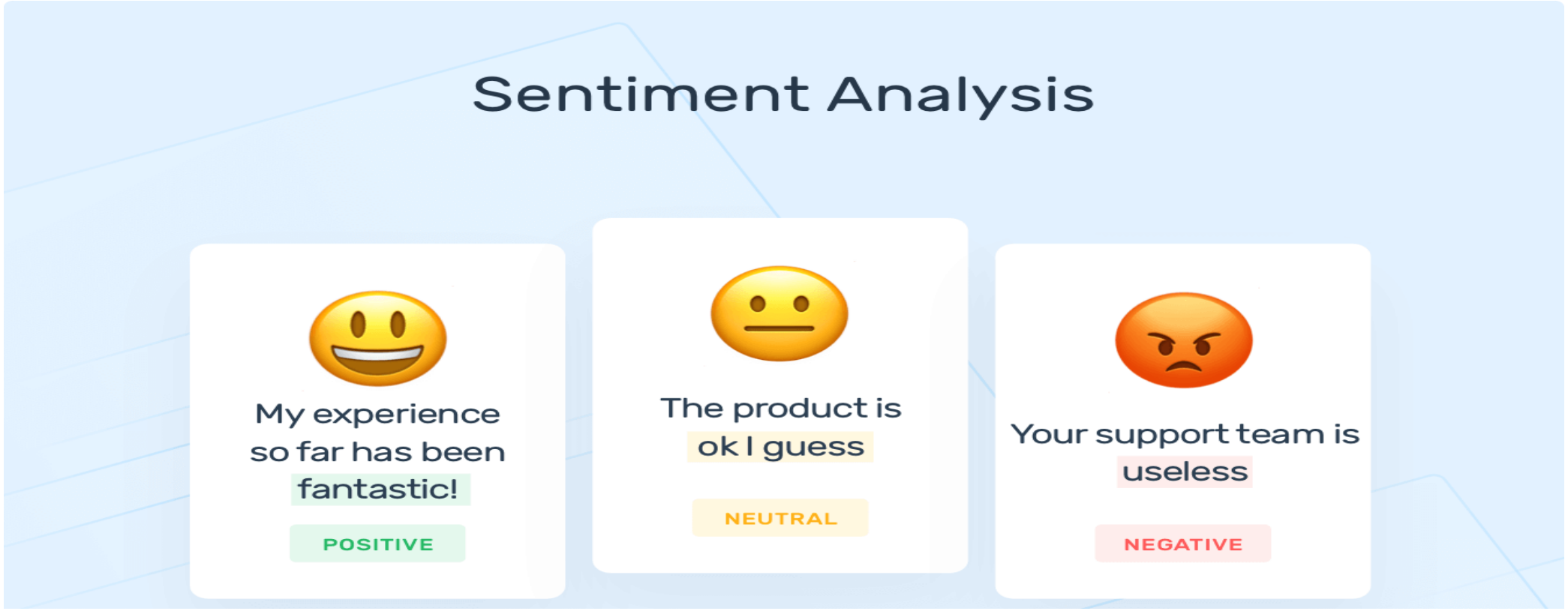
- Any time series problem, like predicting the prices of stocks in a particular month, can be solved using an RNN.



Applications of Recurrent Neural Networks

Natural Language Processing

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



Applications of Recurrent Neural Networks

Machine Translation

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

Advantages of Recurrent Neural Network

- Ability To Handle Variable-Length Sequences
- Memory Of Past Inputs
- Parameter Sharing
- Non-Linear Mapping
- Sequential Processing
- Flexibility
- Improved Accuracy

Disadvantages of Recurrent Neural Network

- Vanishing And Exploding Gradients
- Computational Complexity
- Difficulty In Capturing Long-Term Dependencies
- Lack Of Parallelism
- Difficulty In Choosing The Right Architecture
- Difficulty In Interpreting The Output

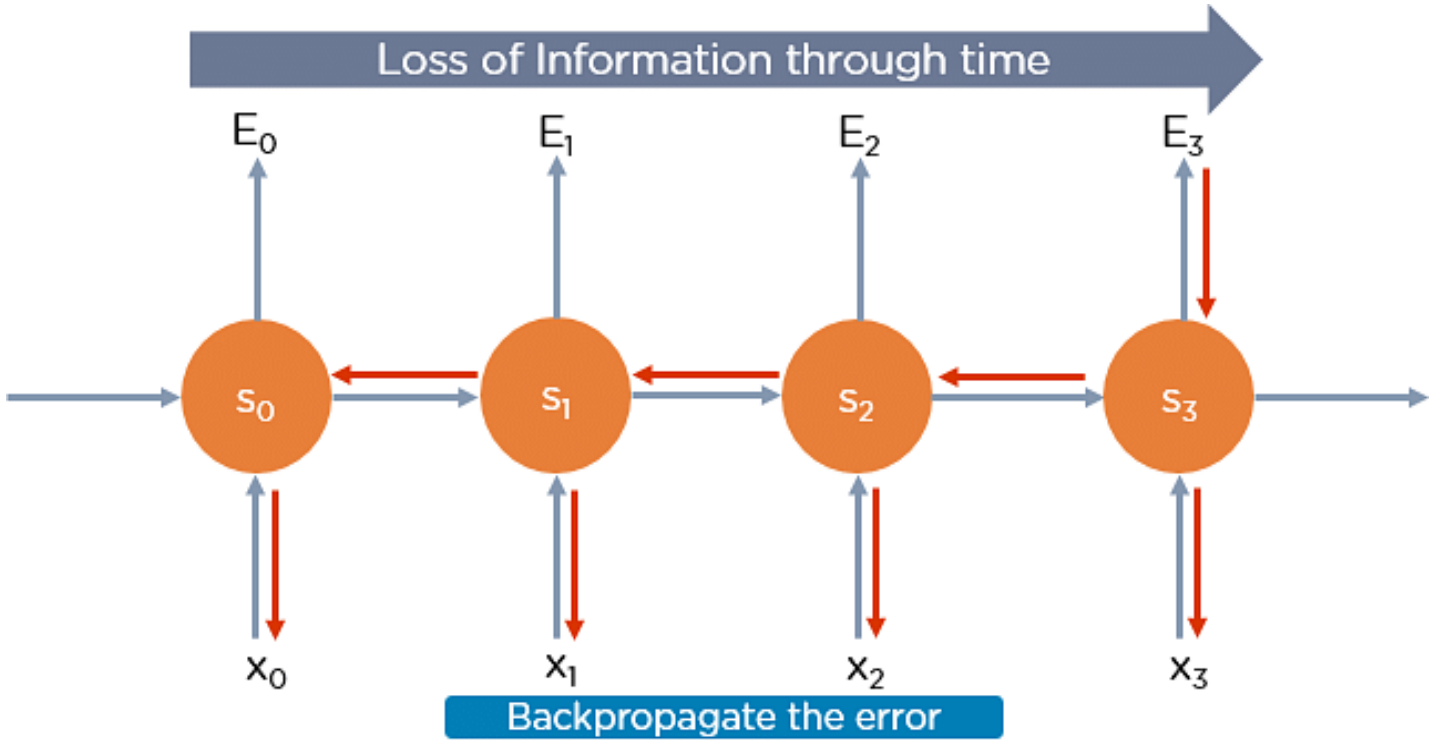
Advantages & Disadvantages of Recurrent Neural Network

- The main **advantage of RNN** over ANN is that **RNN** can model sequence of data (i.e. time series) so that each sample can be assumed to be dependent on previous ones
- Recurrent neural network are even used with convolutional layers to extend the effective pixel neighborhood.
- **Disadvantages**
- Gradient vanishing and exploding problems.
- Training an RNN is a very difficult task.
- It cannot process very long sequences if using *tanh* or *relu* as an activation function.

Two Issues of Standard RNNs

Vanishing Gradient Problem

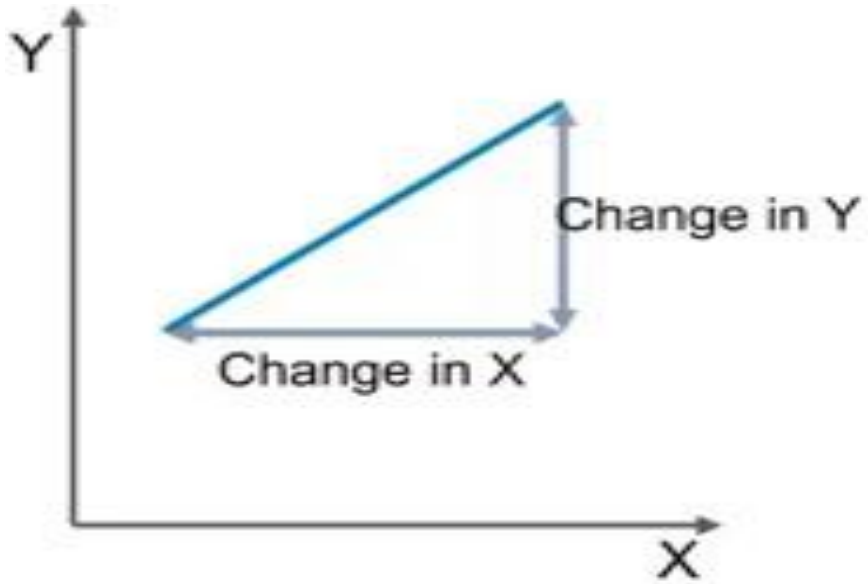
- RNNs suffer from the problem of vanishing gradients.
- The gradients carry information used in the RNN, and when the gradient becomes too small, the parameter updates become insignificant.
- This makes the learning of long data sequences difficult.



Two Issues of Standard RNNs

Exploding Gradient Problem

- While training a neural network, if the slope tends to grow exponentially instead of decaying, this is called an Exploding Gradient. This problem arises when large error gradients accumulate, resulting in very large updates to the neural network model weights during the training process.
- Long training time, poor performance, and bad accuracy are the major issues in gradient problems.



Common Activation Functions

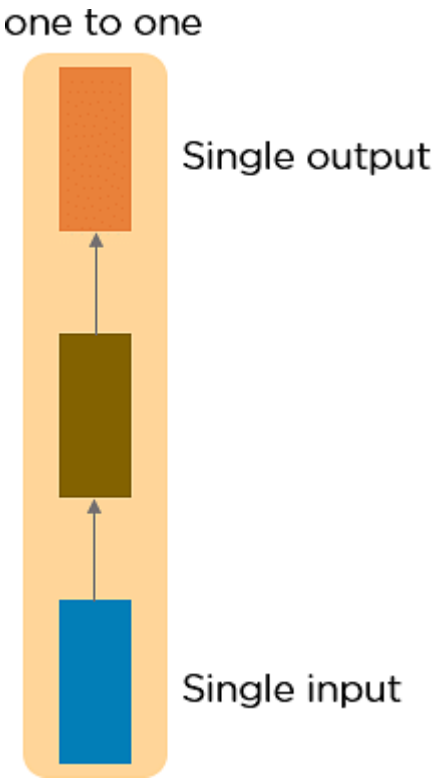
- Sigmoid Function
- Hyperbolic Tangent (Tanh) Function
- Rectified Linear Unit (Relu) Function
- Leaky Relu Function
- Softmax Function

Types of Recurrent Neural Networks

- There are four types of Recurrent Neural Networks:
  1. One to One
  2. One to Many
  3. Many to One
  4. Many to Many

One to One

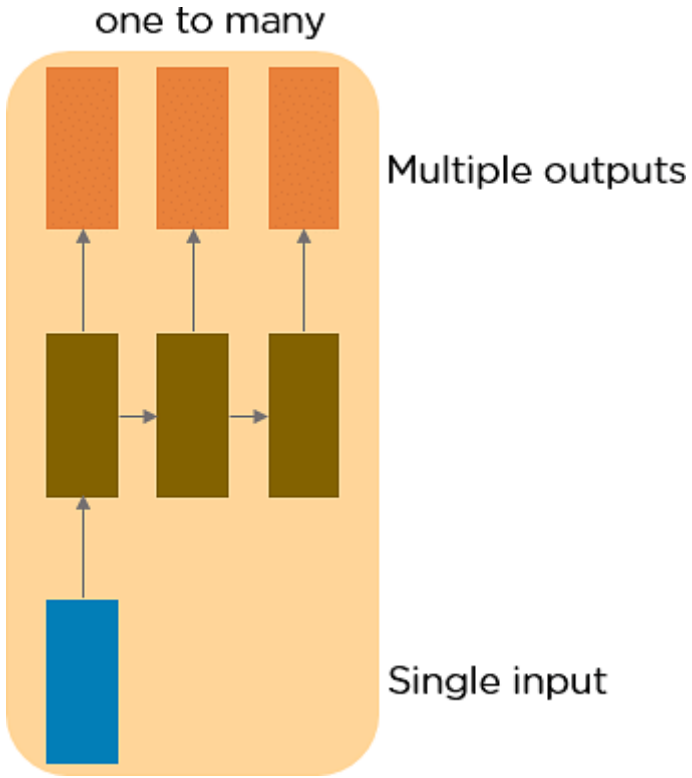
- This type of neural network is known as the Vanilla Neural Network. It's used for general machine learning problems, which has a single input and a single output.



Types of Recurrent Neural Networks

One to Many RNN

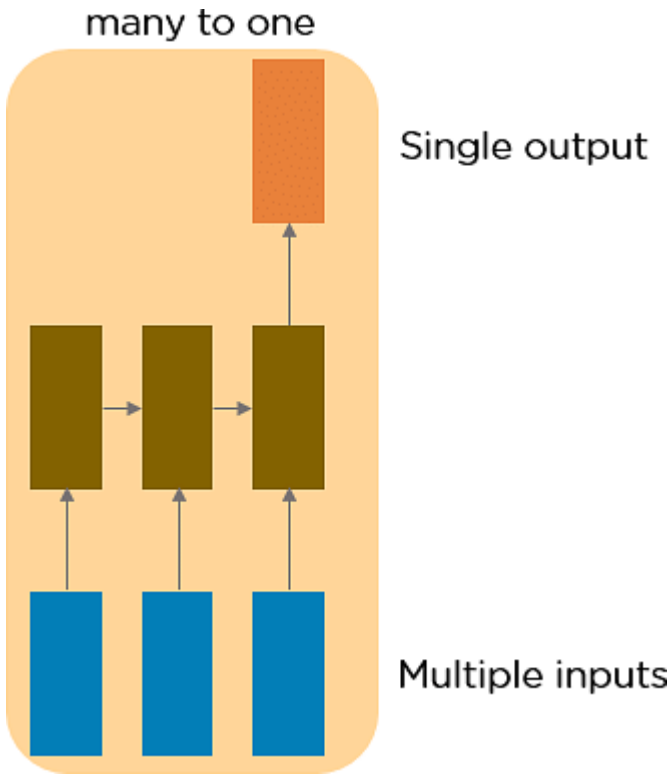
- This type of neural network has a single input and multiple outputs. An example of this is the image caption.



Types of Recurrent Neural Networks

Many to One RNN

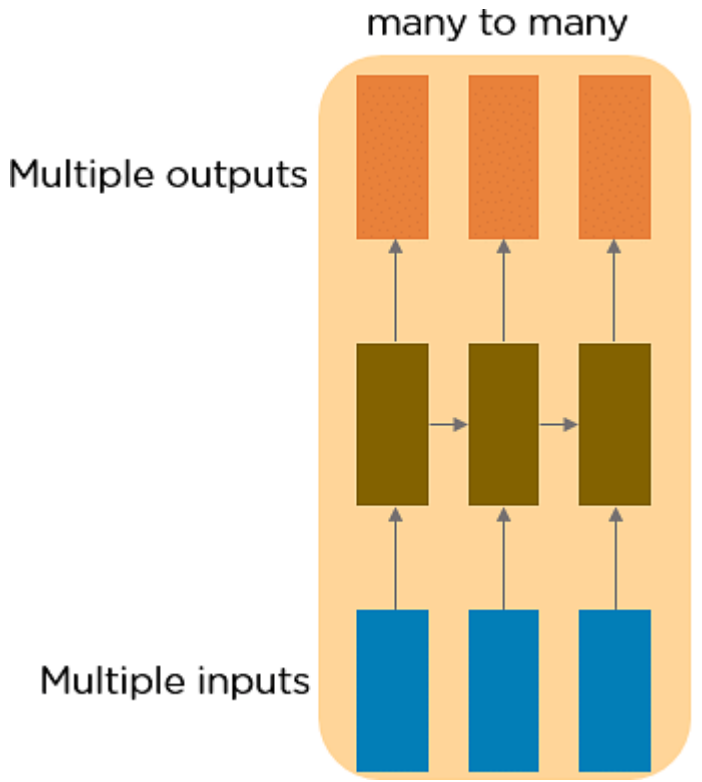
- This RNN takes a sequence of inputs and generates a single output. Sentiment analysis is a good example of this kind of network where a given sentence can be classified as expressing positive or negative sentiments.



Types of Recurrent Neural Networks

Many to Many RNN

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

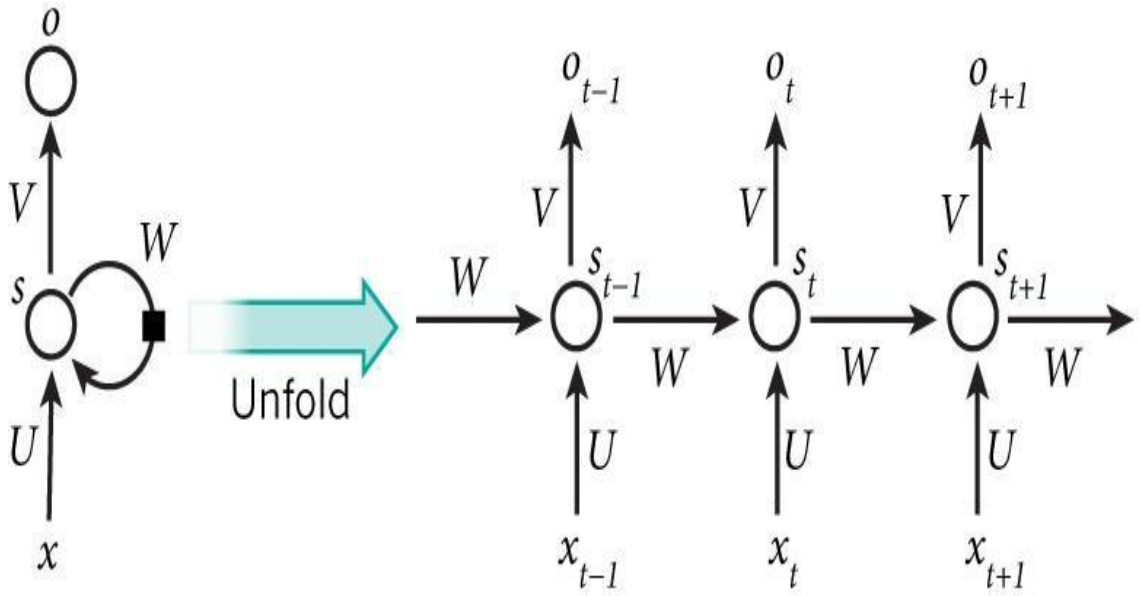


Examples of RNN Flavors

- Vanilla RNN : Fixed size i/p to fixed size o/p (No sequence)
- One to many Example: Image captioning /text generation ()
- Many to 1: Sentiment analysis/DNA classification/music classification
- Many to many : Machine translation (different length)

  : Text summarization
- Many to many(same i/p o/p size) video classification(every frame labeling could be done) Named entity recognition

General Formula



Formula to calculate the current state

$$h_t = f(h_{t-1}, x_t)$$

The equation applying after activation function (tanh) is:

$$h_t = \tanh(w_{hh} h_{t-1} + w_{xh} x_t)$$

- w_{hh} : weight at recurrent neuron,
- W_{xh} : weight at input neuron

Backward propagation in RNN

To train an RNN, we need a loss function. We will make use of cross-entropy loss which is often paired with softmax, which can be calculated as

$$L = -\ln(p_c)$$

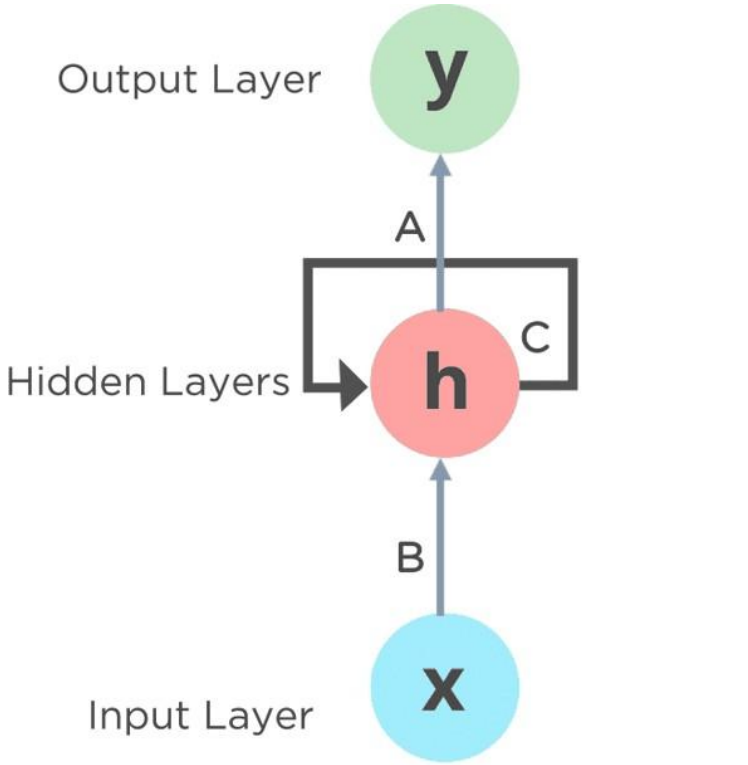
- Here, p_c is the RNN's predicted probability for the correct class (positive or negative). For example, if a positive text is predicted to be 95% positive by the RNN, then the loss is

$$L = -\ln(0.95) = 0.051$$

Recurrent Neural Network(RNN)-Backpropagation through time

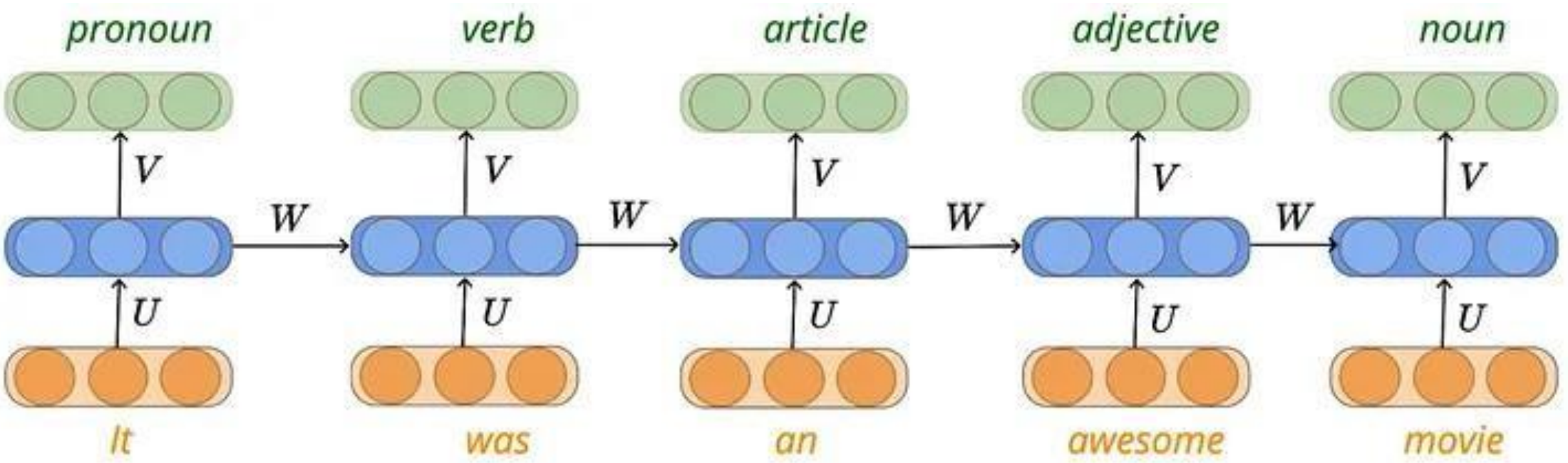
Sample Problems of RNN and BTT

Recap: Working Principle of RNN

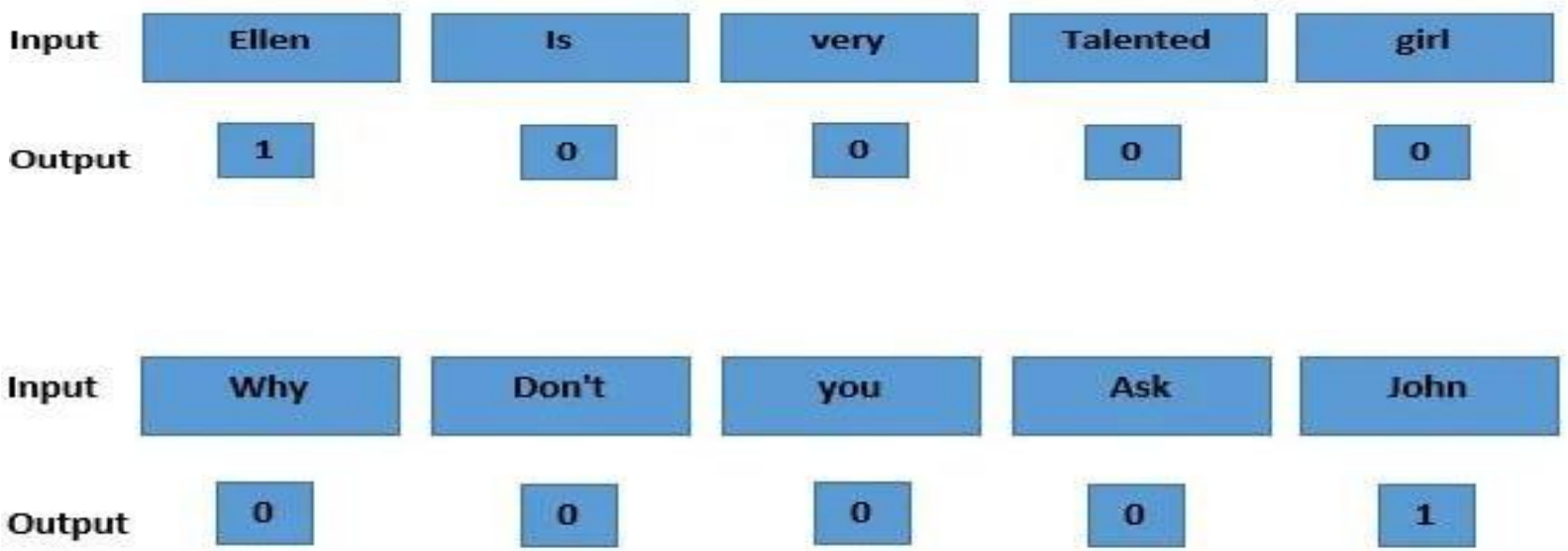


A, B and C are the parameters

Example


pronoun
verb
article
adjective
noun
V
W
U
It
was
an
awesome
movie

- let's take an example of NLP application Named entity recognition, this technique is used to detect names in a sentence :



- for each instance of training (sentence) map each word with an output, if the word is named (john, Ellen …) map it to 1. Otherwise, we map it to 0.

RNN - Example

- It is used in sentiment analysis
- For example, let us take a few sentences (positive and negative)
- These sentences will be converted into numbers(vectors)
- For example (if gender = male,female,transgender) we can use one hot encoding to convert it into 001,010,100….
- This representation has to be given into the network ->>>????
- Word Embedding is one such technique where we can represent the text using vectors.
- The more popular forms of word embeddings are:

1. BoW, which stands for Bag of Words
2. TF-IDF, which stands for Term Frequency-Inverse Document Frequency

Solution 1: Bag of words (BoW)

- One way of representation :
- R1: This movie is very scary and long
- R2: This movie is long and is not slow
- R3: This movie is long spooky good

	1 This	2 movie	3 is	4 very	5 scary	6 and	7 long	8 not	9 slow	10 spooky	11 good	Length of the review(in words)
Review 1	1	1	1	1	1	1	1	0	0	0	0	7
Review 2	1	1	2	0	0	1	1	0	1	0	0	8
Review 3	1	1	1	0	0	0	1	0	0	1	1	6

- Vector of Review 1: [1 1 1 1 1 1 1 0 0 0 0]
- Vector of Review 2: [1 1 2 0 0 1 1 0 1 0 0]
- Vector of Review 3: [1 1 1 0 0 0 1 0 0 1 1]

Shortcomings of BoW

- If the new sentences contain new words, then our vocabulary size would increase and thereby, the length of the vectors would increase too.
- Additionally, the vectors would also contain many 0s, thereby resulting in a sparse matrix (which is what we would like to avoid)
- No information on the grammar of the sentences nor on the ordering of the words in the text is retained.

2. Term Frequency: Inverse Document Frequency(tf-idf)

- Term Frequency (tf): gives us the frequency of the word in each document in the corpus.
- It is the ratio of a number of times the word appears in a document compared to the total number of words in that document.
- It increases as the number of occurrences of that word within the document increases.

$$tf_{i,j} = \frac{n_{i,j}}{\sum_k n_{i,j}}$$

- **Inverse Data Frequency (idf):** used to calculate the weight of rare words across all documents in the corpus.
- The words that occur rarely in the corpus have a high IDF score.

$$idf(w) = log(\frac{N}{df_t})$$

- Combining these two we come up with the TF-IDF score (w) for a word in a document in the corpus. It is the product of tf and idf.

$$w_{i,j} = tf_{i,j} \times \log\left(\frac{N}{df_i}\right)$$

tf_{ij} = number of occurrences of i in j
df_i = number of documents containing i
N = total number of documents

Example

- S1:Inflation has increased unemployment
- S2:The company has increased its sales
- S3: Fear increased his pulse

- **Step 1: Data Pre-processing**

S.No.	Sentences
1.	inflation increased unemployment
2.	company increased sales
3.	fear increased pulse

- **Step 2: Calculating Term Frequency**

Words	Sentences		
	inflation increased unemployment	company increased sales	fear increased pulse
inflation	1/3	0/3	0/3
company	0/3	1/3	0/3
increased	1/3	1/3	1/3
sales	0/3	1/3	0/3
fear	0/3	0/3	1/3
pulse	0/3	0/3	1/3
unemployment	1/3	0/3	0/3

- **Step 3:** Calculating Inverse Document Frequency

Words	Inverse Document Frequency (IDF)
inflation	log(3/1) = 0.477
company	log(3/1) = 0.477
increased	log(3/3) = 0
sales	log(3/1) = 0.477
fear	log(3/1) = 0.477
pulse	log(3/1) = 0.477
unemployment	log(3/1)=0.477

- **Step 4:** Calculating Product of Term Frequency & Inverse Document Frequency

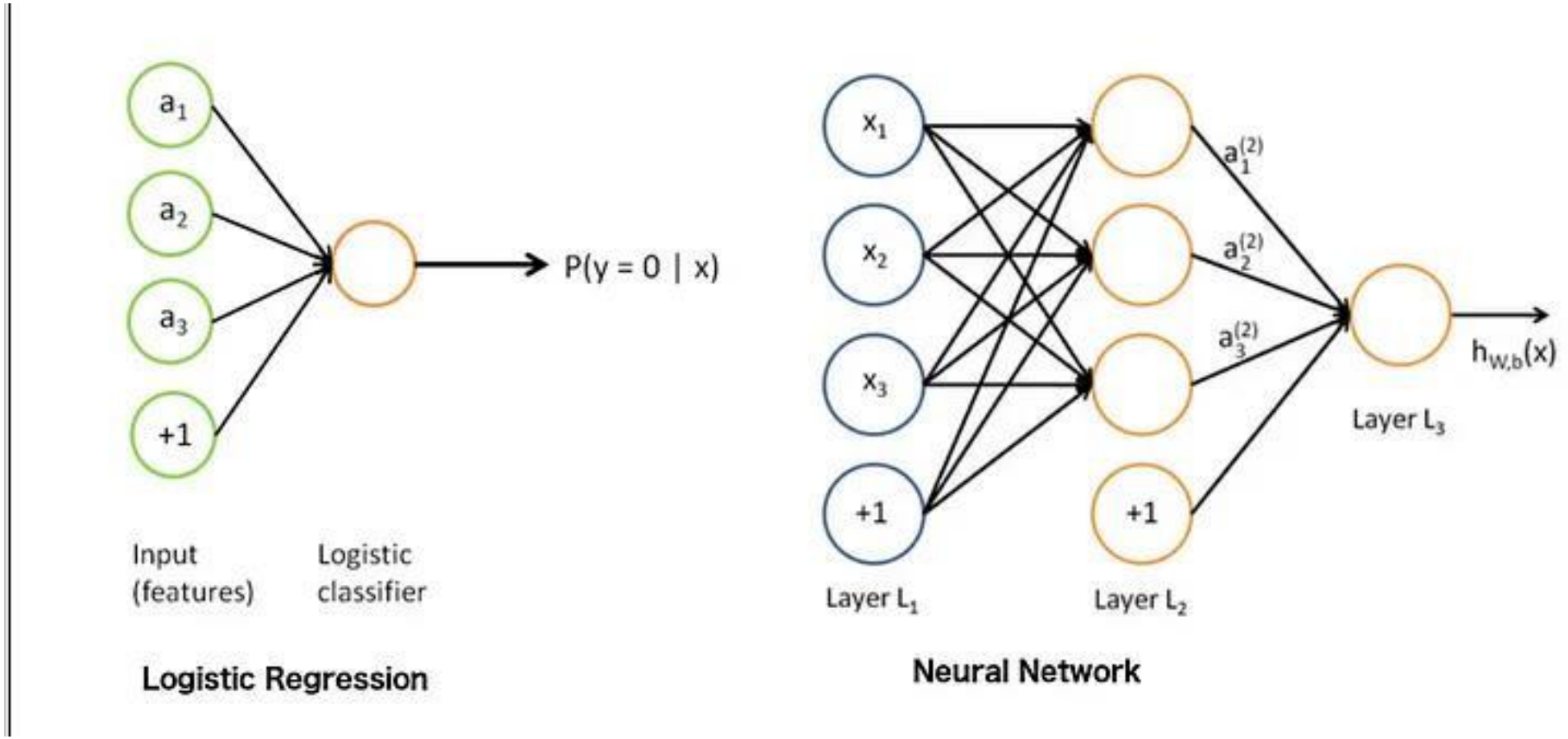
	inflation	company	increased	sales	fear	pulse	unemployment
inflation increased unemployment	1/3*0.477	0/3*0.477	1/3*0	0/3*0.477	0/3*0.477	0/3*0.477	1/3*0.477
company increased sales	0/3*0.477	1/3*0.477	1/3*0	1/3*0.477	0/3*0.477	0/3*0.477	0/3*0.477
fear increased pulse	0/3*0.477	0/3*0.477	1/3*0	0/3*0.477	1/3*0.477	1/3*0.477	0/3*0.477

- Final TF-IDF matrix

	inflation	company	increased	sales	fear	pulse	unemployment
inflation increased unemployment	0.159	0	0	0	0	0	0.159
company increased sales	0	0.159	0	0.159	0	0	0
fear increased pulse	0	0	0	0	0.159	0.159	0

Drawback

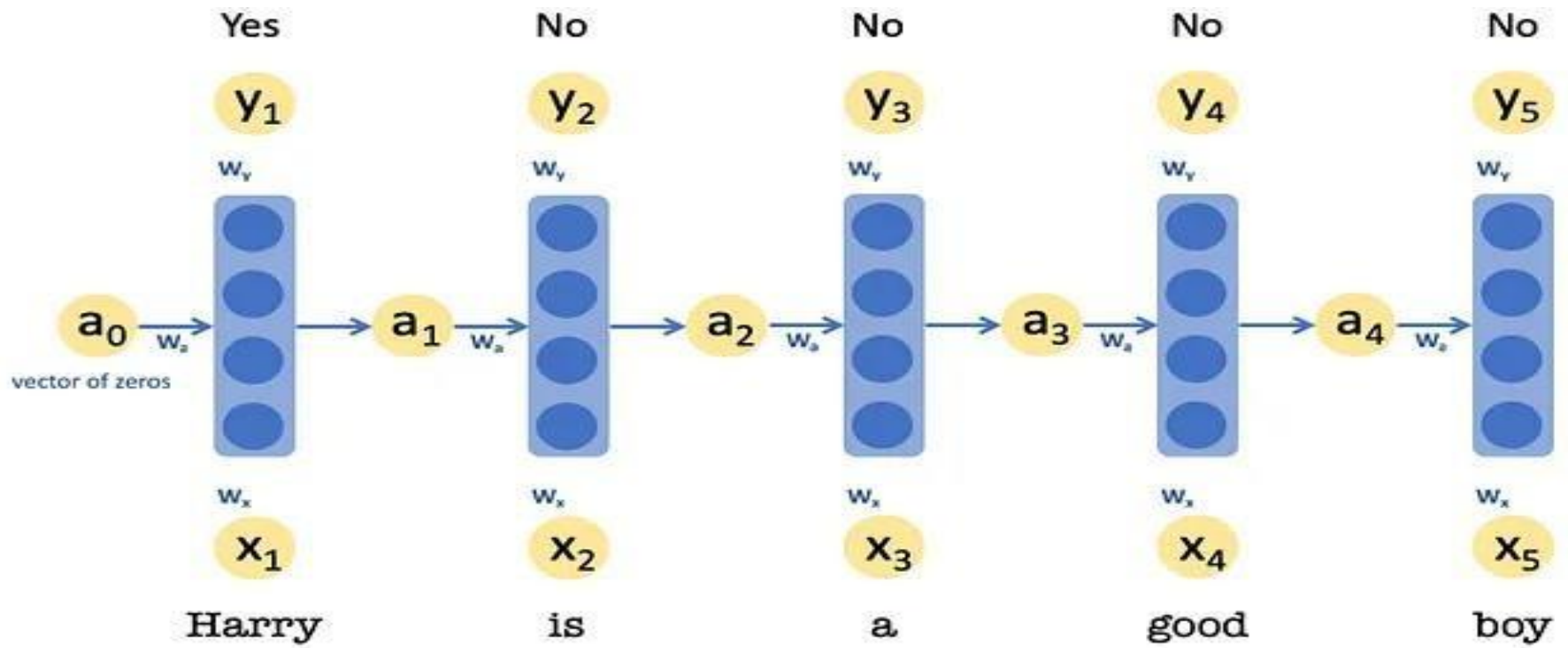
- Feed forward network with BoW does not consider the position of the words in the input.



Sequence Applications

- Music generation
- Speech recognition
- Sentiment classification
- Machine translation
- Activity recognition
- Finding a specific protein from a DNA sequence
- Named Entity Recognition

- To train RNN on sentences to recognize names within, the RNN architecture would be something like that
  - Named entity Recognition



How is the data present in the form of text fed as input to such a neural network model?

- In one hot encoding, every word (even symbols) which are part of the given text data are written in the form of vectors, constituting only of 1 and 0 .
- So one hot vector is a vector whose elements are only 1 and 0.
- Each word is written or encoded as one hot vector, with each one hot vector being unique.
- One word is represented as a vector therefore the list of words in the sentence can be represented as an array of vectors or a matrix
- If there are more words, then one hot vector size will be more

Example of one hot vector

- Increase in size of one hot vector is a great drawback
- If 2 words are similar, but in this technique, similarity will be 0.

The cat sat on the mat

The: [0 1 0 0 0 0 0]

cat: [0 0 1 0 0 0 0]

sat: [0 0 0 1 0 0 0]

on: [0 0 0 0 1 0 0]

the: [0 0 0 0 0 1 0]

mat: [0 0 0 0 0 0 1]

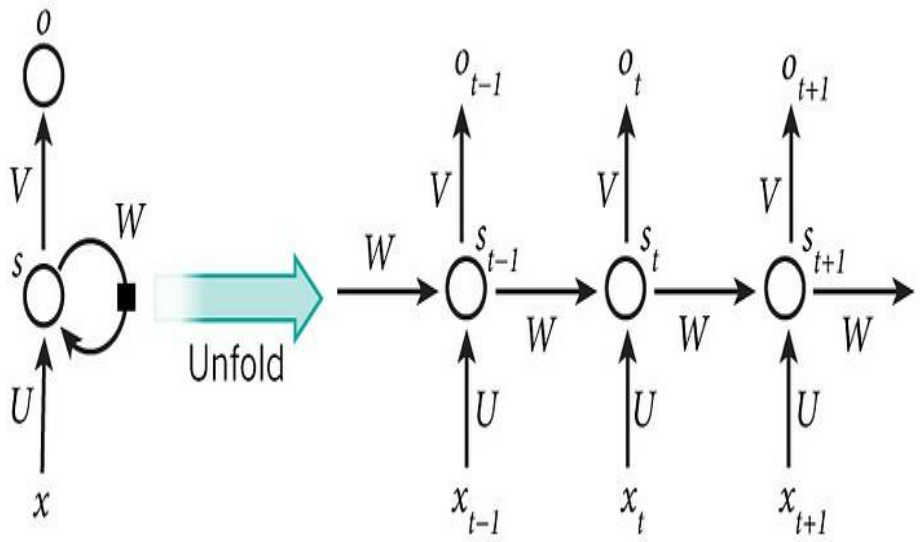
Drawbacks of Standard Neural Network

- Standard neural architecture will not perform well for sequence models
- Feed fwd network accepts a fixed size vector as the input and produces a fixed size vector as the output
- Does not share the features learned across the different positions of the text
- Sequence and length has to be maintained in a network for further processing.

Solution For Sequence Analysis

- Number of parameters for learning is reduced
- Sequence of vectors is processed
- Sharing of parameters across all steps
- Presence of memory
- Previous output is used with the current input
- RNN is not a feed fwd network as cycle is formed in the hidden units

Notations



Input: $x(t)$ is taken as the input to the network at time step t. For example, $x1$,could be a one-hot vector corresponding to a word of a sentence.

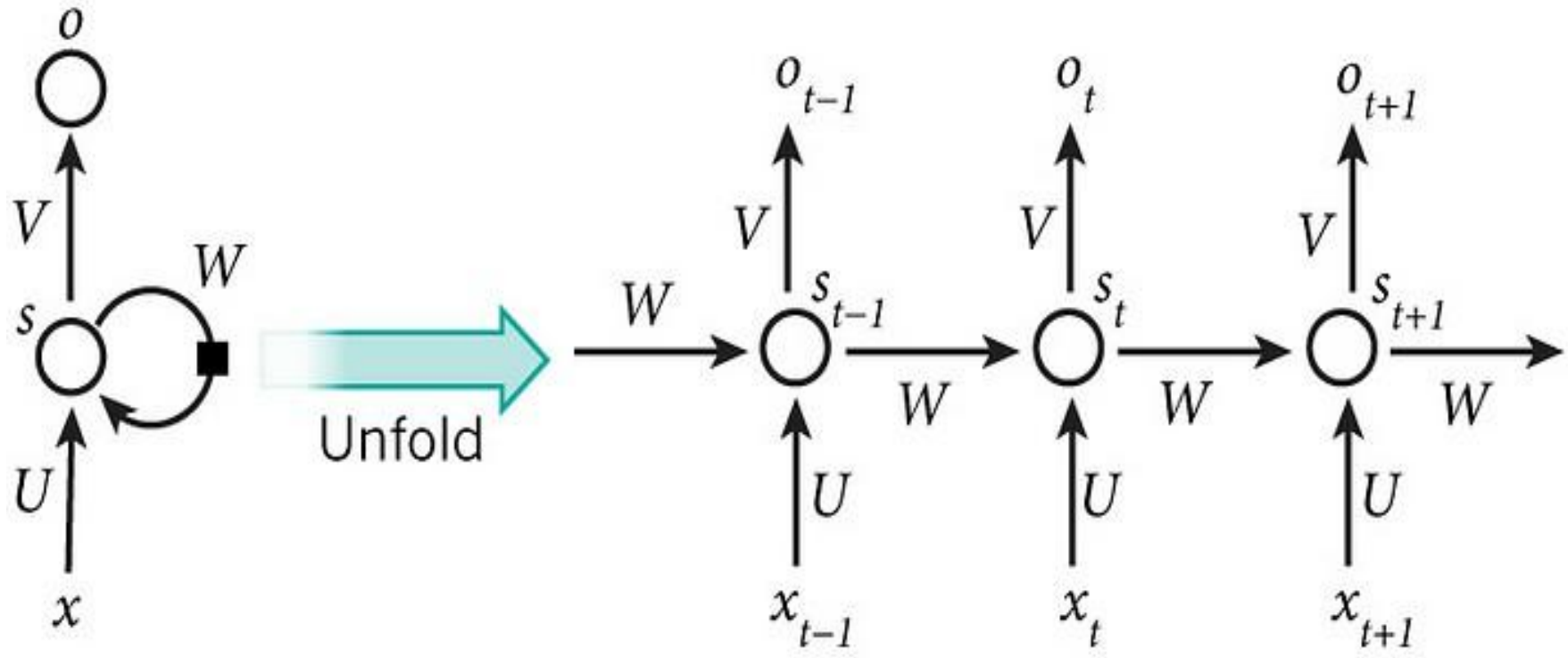
Hidden state: h(t) represents a hidden state at time t and acts as "memory" of the network. h(t) is calculated based on the current input and the previous time step's hidden state $\mathbf{h(t)} = f(U\,\mathbf{x}^{(t)} + W\,\mathbf{h}^{(t-1)})$.
The function f is taken to be a non-linear transformation such as tanh, ReLU.

Weights: The RNN has input to hidden connections parameterized by a weight matrix U, hidden-to-hidden recurrent connections parameterized by a weight matrix W, and hidden-to-output connections parameterized by a weight matrix V and all these weights (U,V,W) are shared across time.

Output: o(t) illustrates the output of the network. In the figure, I just put an arrow after o(t) which is also often subjected to non-linearity, especially when the network contains further layers downstream.

RNN - Forward Pass

An **RNN (Recurrent Neural Network)** processes data sequentially, meaning it takes one input at a time and remembers past information using a hidden state.



1. **Understanding the Hidden State**
    - RNN has a memory, called the **hidden state**, which helps it remember past inputs.
    - At each step, the RNN updates this hidden state using both the **current input** and the **previous hidden state**.

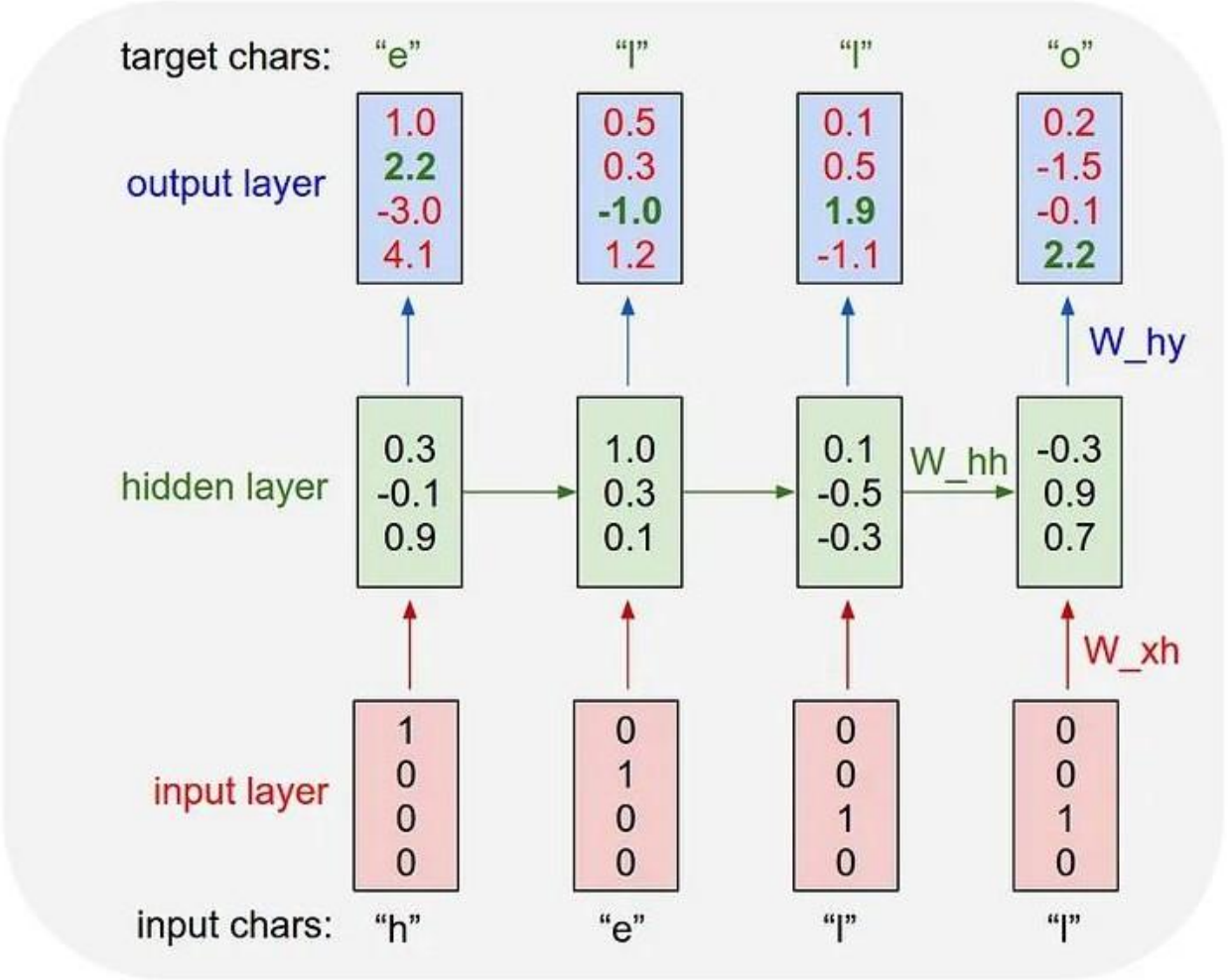
2. **Processing Input and Generating Output**
    - The input is **one-hot encoded** (converted into a numerical format where each character has its own unique representation).
    - The input is multiplied by a **weight matrix (U)** to transform it into a different representation.
    - The hidden state is updated using the transformed input and the previous hidden state.
    - The final output is generated using another weight matrix and passed through a **softmax function** to predict the next character.

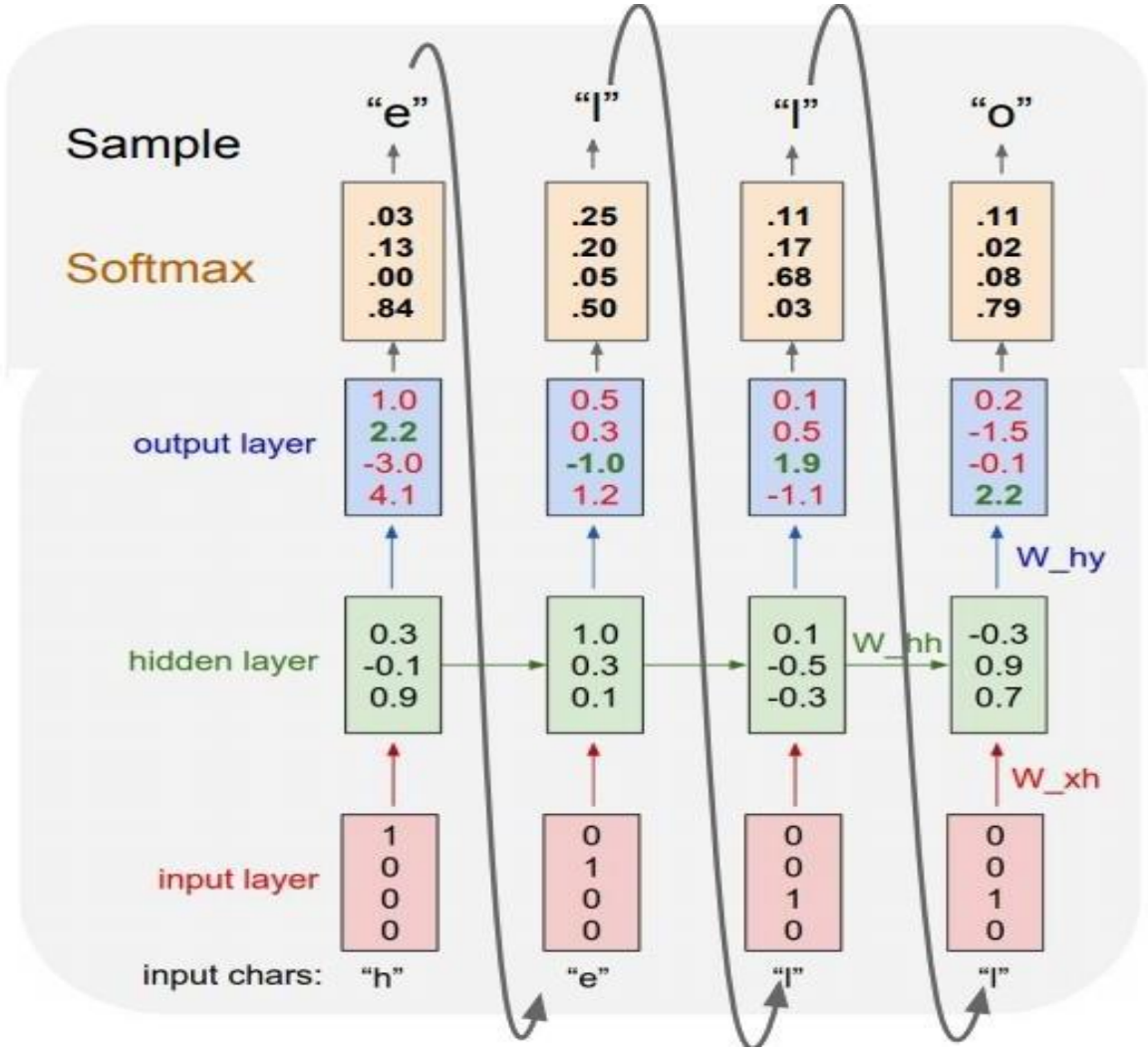
3. **Example - Predicting Characters**
    - Suppose we train the RNN on the word "**hello**".
    - If we give the model "**h**", it should predict "**e**".
    - If we give "**e**", it should predict "**l**", and so on.
4. **Weights and Initialization**
    - The weight matrix **U** is used to transform the one-hot encoded input.
    - This matrix is randomly initialized at the start and gets updated during training.

The RNN learns patterns in sequential data by updating its memory (hidden state) at each step, allowing it to predict the next item in a sequence.

Example

- For example, if we train on the string ‘hello’, then feed in ‘h’ to the model it should predict ‘e’; if we feed in ‘e’, it should predict ’l’, and so on





Current hidden state s at time t will be:

$$o^{(t)} = c + Vh^{(t)}$$

$$\hat{y}^{(t)} = \text{softmax}(o^{(t)})$$

1. Computing Raw Output ($o^{(t)}$)

This step transforms the hidden state into a meaningful representation for prediction.

- $o^{(t)}$ is an intermediate value used before generating the final prediction.
- It is computed as:

$$o^{(t)} = c + Vh^{(t)}$$

  - $h^{(t)}$ → Hidden state at time t (memory).
  - V → Weight matrix that transforms the hidden state into output space.
  - c → Bias term that helps adjust the output.

2. Generating Final Output ($\hat{y}^{(t)}$)

$$o^{(t)} = c + Vh^{(t)}$$

$$\hat{y}^{(t)} = \text{softmax}(o^{(t)})$$

Softmax makes the output interpretable as a probability distribution.

- The raw output $o^{(t)}$ is passed through a **softmax** function:

$$\hat{y}^{(t)} = \text{softmax}(o^{(t)})$$

  - **Softmax** converts $o^{(t)}$ into probabilities, ensuring the output represents a valid prediction (e.g., probabilities of different characters in a sequence).

1. Compute raw output: $o^{(t)} = c + Vh^{(t)}$
2. Convert to probabilities: $\hat{y}^{(t)} = \text{softmax}(o^{(t)})$

Input x at a given time t will be:

$$a^{(t)} = b + Wh^{(t-1)} + Ux^{(t)}$$

$$h^{(t)} = \tanh(a^{(t)})$$

1. Computing Intermediate Activation ($a^{(t)}$)

This equation **combines past memory and current input** before applying an activation function.

- Before updating the hidden state, we compute an intermediate value:

$$a^{(t)} = b + Wh^{(t-1)} + Ux^{(t)}$$

  - $h^{(t-1)}$ → Hidden state from the previous time step (memory).
  - $x^{(t)}$ → Input at time t (e.g., a character in a sequence).
  - W, U → Weight matrices that determine how past hidden states and current inputs influence the new state.
  - b → Bias term that helps adjust the activation.

$$a^{(t)} = b + Wh^{(t-1)} + Ux^{(t)}$$

$$h^{(t)} = \tanh(a^{(t)})$$

2. Computing the Updated Hidden State ($h^{(t)}$)

This step updates the **hidden state**, which carries memory for future time steps.

- The intermediate activation $a^{(t)}$ is passed through a **tanh** function:

$$h^{(t)} = \tanh(a^{(t)})$$

- **Tanh** squashes the values between **-1 and 1**, helping the network learn in a stable way.

This process allows the RNN to **integrate new inputs with past memory** for sequential learning!

1. Compute intermediate activation: $a^{(t)} = b + Wh^{(t-1)} + Ux^{(t)}$
2. Update hidden state: $h^{(t)} = \tanh(a^{(t)})$

Current output o at time t will be computing using

$$o^{(t)} = c + Vh^{(t)}$$

$$\hat{y}^{(t)} = \text{softmax}(o^{(t)})$$

1. Computing Raw Output ($o^{(t)}$)

This equation **transforms the hidden state** into a form suitable for prediction.

- Before generating the final prediction, an intermediate value is computed:

$$o^{(t)} = c + Vh^{(t)}$$

  - $h^{(t)}$ → Hidden state at time t (memory of previous inputs).
  - V → Weight matrix that transforms the hidden state into output space.
  - c → Bias term that helps adjust the output.

2. Generating Final Prediction ($\hat{y}^{(t)}$)

$$o^{(t)} = c + Vh^{(t)}$$

$$\hat{y}^{(t)} = \text{softmax}(o^{(t)})$$

Softmax makes the output **interpretable as a probability distribution**, allowing the model to make a prediction

- The raw output $o^{(t)}$ is passed through a **softmax** function:

$$\hat{y}^{(t)} = \text{softmax}(o^{(t)})$$

  - **Softmax** converts $o^{(t)}$ into probabilities, ensuring the output represents a valid prediction (e.g., probabilities of different characters or words in a sequence).

1. Compute raw output: $o^{(t)} = c + Vh^{(t)}$
2. Convert to probabilities: $\hat{y}^{(t)} = \text{softmax}(o^{(t)})$

RNN - Forward Pass Example

Suppose we train the RNN on the word "**hello**".

Step 1: One-hot Encoding and Weight Initialization

- The input vocabulary consists of (**h, e, l, o**).
- Each letter is one-hot encoded.
  - $h = [1, 0, 0]$
  - $e = [0, 1, 0]$
  - $l = [0, 0, 1]$

Note:

The output will not necessarily be generated for every t. It depends upon the application.

Ex: In speech recognition, RNN will output words instantly at every iteration.

Opinion classification RNN will provide output only at the end of the sentence.

- Using weight, the one-hot encoding vectors are transformed using the weight 'U'.
- Random initialization of weights 3* 4 matrix :
  - The weight matrix U (randomly initialized) transforms these one-hot vectors:

$$U = \begin{bmatrix} 0.28231 & 0.34343 & 0.68823 & 0.48642 \\ 0.98333 & 0.45434 & 0.37973 & 0.12368 \\ 0.53752 & 0.32986 & 0.23499 & 0.86723 \end{bmatrix}$$

Step 2: Compute Hidden State for 'h'

- For the letter '**h**', the one-hot encoding is **(1, 0, 0)**.
- The transformed representation is:

$$U \times X_h = \begin{bmatrix} 0.28231 \\ 0.98333 \\ 0.53752 \end{bmatrix}$$

- The recurrent weight matrix W and bias b are given as:

$$W = 0.427043, \quad b = 0.56702$$

- Since '**h**' is the first character, there's no previous hidden state:

$$W \times S_{t-1} + b = 0 + 0.56702 = 0.56702$$

- Final hidden state before activation:

$$U \times X_h + b = \begin{bmatrix} 0.28231 \\ 0.98333 \\ 0.53752 \end{bmatrix} + 0.56702 = \begin{bmatrix} 0.84933 \\ 1.55035 \\ 1.10454 \end{bmatrix}$$

- Apply **tanh** activation:

$$S_t = \tanh(UX_h + b) = \begin{bmatrix} 0.6907 \\ 0.9138 \\ 0.80212 \end{bmatrix}$$

We assume the hyperbolic tangent activation function for the hidden layer.

Step 3: Compute Hidden State for 'e'

- The one-hot encoding for 'e' is (0, 1, 0).
- The transformed representation:

$$U \times X_e = \begin{bmatrix} 0.34343 \\ 0.45434 \\ 0.32986 \end{bmatrix}$$

- Compute the new hidden state:

$$W \times S_t + b = 0.427043 \begin{bmatrix} 0.6907 \\ 0.9138 \\ 0.80212 \end{bmatrix} + 0.56702$$

$$= \begin{bmatrix} 0.86197 \\ 0.95725 \\ 0.90955 \end{bmatrix}$$

- Final hidden state before activation:

$$\begin{bmatrix} 0.86197 \\ 0.95725 \\ 0.90955 \end{bmatrix} + \begin{bmatrix} 0.34343 \\ 0.45434 \\ 0.32986 \end{bmatrix} = \begin{bmatrix} 1.2054 \\ 1.41159 \\ 1.23941 \end{bmatrix}$$

- Apply tanh:

$$S_t = \begin{bmatrix} 0.93653 \\ 0.94910 \\ 0.76234 \end{bmatrix}$$

Step 3: Compute Hidden State for 'l'

- The one-hot encoding for 'l' is (0, 0, 1).
- The transformed representation:

$$U \times X_l = \begin{bmatrix} 0.68823 \\ 0.37973 \\ 0.23499 \end{bmatrix}$$

- The hidden state from 'e' was:

$$S_e = \begin{bmatrix} 0.93653 \\ 0.94910 \\ 0.76234 \end{bmatrix}$$

- Compute the new hidden state:

$$W \times S_e + b = 0.427043 \begin{bmatrix} 0.93653 \\ 0.94910 \\ 0.76234 \end{bmatrix} + 0.56702$$

$$= \begin{bmatrix} 0.96884 \\ 0.97426 \\ 0.89278 \end{bmatrix}$$

- Final hidden state before activation:

$$\begin{bmatrix} 0.96884 \\ 0.97426 \\ 0.89278 \end{bmatrix} + \begin{bmatrix} 0.68823 \\ 0.37973 \\ 0.23499 \end{bmatrix} = \begin{bmatrix} 1.65707 \\ 1.35399 \\ 1.12777 \end{bmatrix}$$

- Apply tanh activation:

$$S_l = \begin{bmatrix} 0.92975 \\ 0.87444 \\ 0.80993 \end{bmatrix}$$

Step 4: Compute Hidden State for the Second 'I'

- The second 'I' has the same one-hot encoding:

$$X_I = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

- The transformed representation remains:

$$UX_I = \begin{bmatrix} 0.68823 \\ 0.37973 \\ 0.23499 \end{bmatrix}$$

- The previous hidden state (from the first 'I') was:

$$S_I = \begin{bmatrix} 0.92975 \\ 0.87444 \\ 0.80993 \end{bmatrix}$$

- Compute the new hidden state:

$$WS_I + b = 0.427043 \begin{bmatrix} 0.92975 \\ 0.87444 \\ 0.80993 \end{bmatrix} + 0.56702$$

$$= \begin{bmatrix} 0.96485 \\ 0.94158 \\ 0.91343 \end{bmatrix}$$

- Final hidden state before activation:

$$\begin{bmatrix} 0.96485 \\ 0.94158 \\ 0.91343 \end{bmatrix} + \begin{bmatrix} 0.68823 \\ 0.37973 \\ 0.23499 \end{bmatrix} = \begin{bmatrix} 1.65308 \\ 1.32131 \\ 1.14842 \end{bmatrix}$$

- Apply tanh activation:

$$S_{I_2} = \begin{bmatrix} 0.92943 \\ 0.86778 \\ 0.81735 \end{bmatrix}$$

Step 5: Compute Output Y_o for 'o'

- The final character in the sequence is '**o**'.
- **Use the hidden state from the second 'l':**

$$S_{l_2} = \begin{bmatrix} 0.92943 \\ 0.86778 \\ 0.81735 \end{bmatrix}$$

- Compute **raw output before softmax:**

$$Y_o = V S_{l_2}$$

Given:

$$V = \begin{bmatrix} 1.90362 \\ 1.12575 \\ 0.94883 \\ 1.26514 \end{bmatrix}$$

$$V S_{l_2} = \begin{bmatrix} 1.87946 \\ 1.11293 \\ 0.93922 \\ 1.24658 \end{bmatrix}$$

- Apply softmax to get the probability distribution:

$$\text{Softmax}(Y_o) = \begin{bmatrix} 0.4123 \\ 0.1936 \\ 0.1602 \\ 0.2339 \end{bmatrix}$$

The highest probability is **0.4123**, corresponding to a character (which might not be 'o' exactly due to untrained weights).

3 hidden node and 4 o/p nodes since there are 4 letters

- The RNN incorrectly predicts 'h' instead of 'o'.
- This misclassification is due to untrained weights.
- After training with Backpropagation Through Time (BPTT), the model should correctly predict 'o' with the highest probability.

RNN Backpropagation

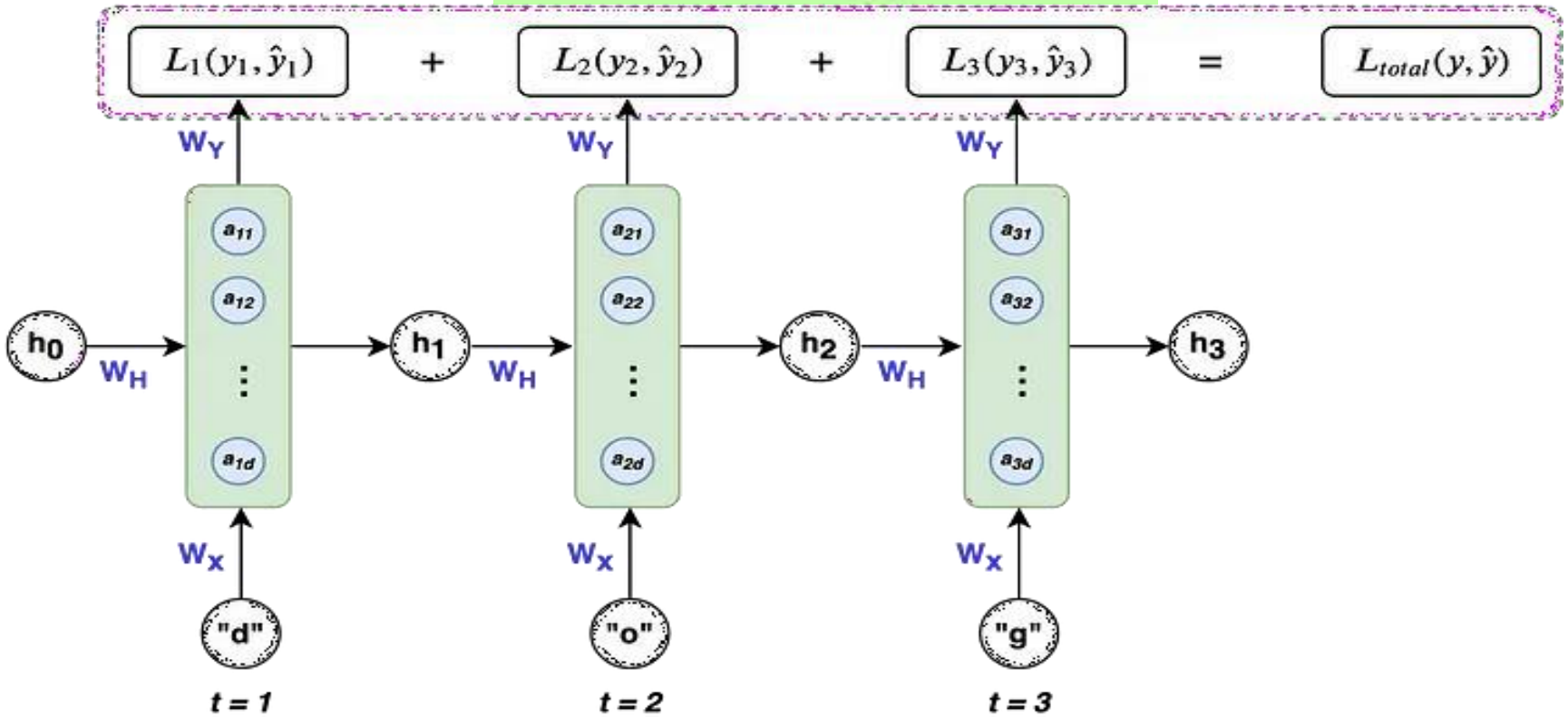
Backpropagation with RNNs is a little more challenging due to the recursive nature of the weights and their effect on the loss which spans over time.

1. Initialize weight matrices *Wx, Wy, Wh* randomly
2. Forward propagation to compute predictions
3. Compute the loss (multi-class cross-entropy loss function for hello example)
4. Backpropagation to compute gradients
5. Update weights based on gradients
6. Repeat steps 2-5

- For example to predict four possible letters ("d-o-g-s"), it makes sense to use the multi-class cross-entropy loss function:

- The overall loss is $L_t(y_t, \hat{y}_t) = -y_t log(\hat{y}_t)$

$$L_{total}(y, \hat{y}) = -\sum_{t=1}^{n} y_t log(\hat{y}_t)$$



How weights are updated?

- We need to calculate the gradients for our three weight matrices *Wx, Wy, Wh*, and update them with a learning rate η.
- Similar to normal backpropagation, the gradient gives us a sense of how the loss is changing with respect to each weight parameter.
- We update the weights to minimize loss with the following equation:

$$W_i := W_i - \eta \frac{\partial L_{total}(y, \hat{y})}{\partial W_i}$$

Step 1: Compute Gradients for Weight Matrices

We have three weight matrices:

1. W_x → Input-to-hidden weights
2. W_h → Hidden-to-hidden weights
3. W_y → Hidden-to-output weights

The gradients we need to compute are:

- $\frac{\partial L}{\partial W_x}$
- $\frac{\partial L}{\partial W_h}$
- $\frac{\partial L}{\partial W_y}$

Since the loss LLL is computed over multiple timesteps / timestamps, we unroll the RNN and apply Backpropagation Through Time (BPTT).

How weights are updated? Cont'd

- The tricky part is to calculate the gradient for *Wx, Wy, and Wh*. We'll start by calculating the gradient for *Wy* because it's the easiest.
- As stated before, the effect of the weights on loss spans over time.
- The weight gradient for *Wy* is the following:

Function dependencies with respect to W_Y

$$L_t = -y_t log(\hat{y}_t) \rightarrow \hat{y}_t = softmax(z_t) \rightarrow z_t = W_Y h_t$$

Chain rule with respect to W_Y

$$\frac{\partial L_t}{\partial W_Y} = \frac{\partial L_t}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial z_t} \frac{\partial z_t}{\partial W_Y}$$

Gradient for W_Y

$$\frac{\partial L_{total}}{\partial W_Y} = \frac{\partial L_1}{\partial \hat{y}_1} \frac{\partial \hat{y}_1}{\partial z_1} \frac{\partial z_1}{\partial W_Y} + \frac{\partial L_2}{\partial \hat{y}_2} \frac{\partial \hat{y}_2}{\partial z_2} \frac{\partial z_2}{\partial W_Y} + \frac{\partial L_3}{\partial \hat{y}_3} \frac{\partial \hat{y}_3}{\partial z_3} \frac{\partial z_3}{\partial W_Y} = \sum_{t=1}^{n} \frac{\partial L_t}{\partial W_Y}$$

How weights are updated? Cont'd

Function Dependencies with respect to W_X

$$L_t = -y_t log(\hat{y}_t) \rightarrow \hat{y}_t = softmax(z_t) \rightarrow z_t = W_Y h_t \rightarrow h_t = tanh(W_H h_{t-1} + W_X X_t) \quad (1)$$

Chain rule with respect to W_x

$$\frac{\partial L_t}{\partial W_Y} = \frac{\partial L_t}{\partial \hat{y}_t}\frac{\partial \hat{y}_t}{\partial z_t}\frac{\partial z_t}{\partial h_t}\frac{\partial h_t}{\partial W_x}$$ but note, that within h_t, h_{t-1} also contains W_x, thus we need to recurisvely chain rule h_{t-1} until we reach h_0 (2)

$$\frac{\partial L_t}{\partial W_Y} = \frac{\partial L_t}{\partial \hat{y}_t}\frac{\partial \hat{y}_t}{\partial z_t}\frac{\partial z_t}{\partial h_t}\frac{\partial h_t}{\partial W_x} + \frac{\partial L_t}{\partial \hat{y}_t}\frac{\partial \hat{y}_t}{\partial z_t}\frac{\partial z_t}{\partial h_t}\frac{\partial h_t}{\partial h_{t-1}}\frac{\partial h_{t-1}}{\partial W_x} + \ldots + \frac{\partial L_t}{\partial \hat{y}_t}\frac{\partial \hat{y}_t}{\partial z_t}\frac{\partial z_t}{\partial h_t}\frac{\partial h_t}{\partial h_{t-n}}\frac{\partial h_{t-n}}{\partial W_x} = \sum_{k=0}^{n}\frac{\partial L_t}{\partial \hat{y}_t}\frac{\partial \hat{y}_t}{\partial z_t}\frac{\partial z_t}{\partial h_t}\frac{\partial h_t}{\partial h_k}\frac{\partial h_k}{\partial W_x} \quad (3)$$

Gradient for W_X

$$\frac{\partial L_{total}}{\partial W_X} = \sum_{t=1}^{n}\sum_{k=0}^{n}\frac{\partial L_t}{\partial \hat{y}_t}\frac{\partial \hat{y}_t}{\partial z_t}\frac{\partial z_t}{\partial h_t}\frac{\partial h_t}{\partial h_k}\frac{\partial h_k}{\partial W_x} \quad (4)$$

BP through time (BPTT)

- Backpropagation Through Time (BPTT) is the method used to train RNNs by adjusting their weights to minimize errors.
- BPTT works by unrolling all input timesteps. Each timestep has one input timestep, one copy of the network, and one output.
- Errors are then calculated and accumulated for each timestep.
- The network is rolled back up and the weights are updated.

How Does BPTT Work?

1. **Unrolling the Network:**
    - Instead of treating the RNN as a single entity, we **expand** it over time.
    - Each time step has its **own input**, **hidden state**, and **output**.
    - For example, if the input is `"hello"`, the network processes one letter at a time while maintaining memory.

2. **Forward Pass:**
    - The network computes predictions **step by step**.
    - Each hidden state depends on the **previous hidden state**.
    - The final output is compared with the actual output to compute the **loss**.

3. **Backward Pass (Error Calculation & Weight Update):**

- The error is computed at the last time step.
- The error is **propagated backward** through each previous time step.
- Gradients are accumulated over time to update the weights W_x, W_h, W_y.
- The network is **rolled back up**, and weights are adjusted based on all time steps.

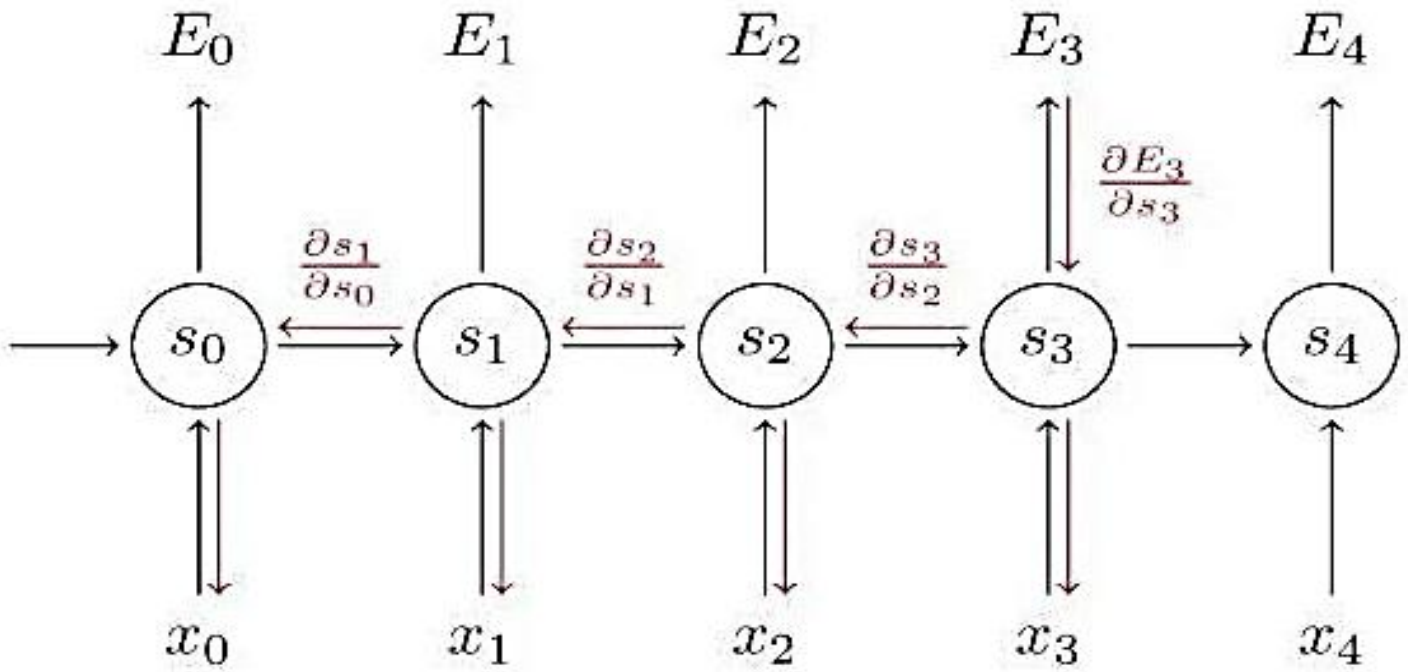
The total error gradient with respect to the weights is the **sum of all individual time step errors**.

$$\frac{\partial E}{\partial W} = \sum_t \frac{\partial E_t}{\partial W}$$

Summation of Gradients Over Time Steps

$$\frac{\partial E_3}{\partial W} = \sum_{k=0}^{3} \frac{\partial E_3}{\partial \hat{y}_3} \cdot \frac{\partial \hat{y}_3}{\partial s_3} \cdot \frac{\partial s_3}{\partial s_k} \cdot \frac{\partial s_k}{\partial W}$$

- BPTT unrolls the RNN across all timesteps.
- Errors are backpropagated step by step.
- Each hidden state depends on previous ones, creating a chain of dependencies.
- Weights are updated based on the accumulated gradients over all time steps.

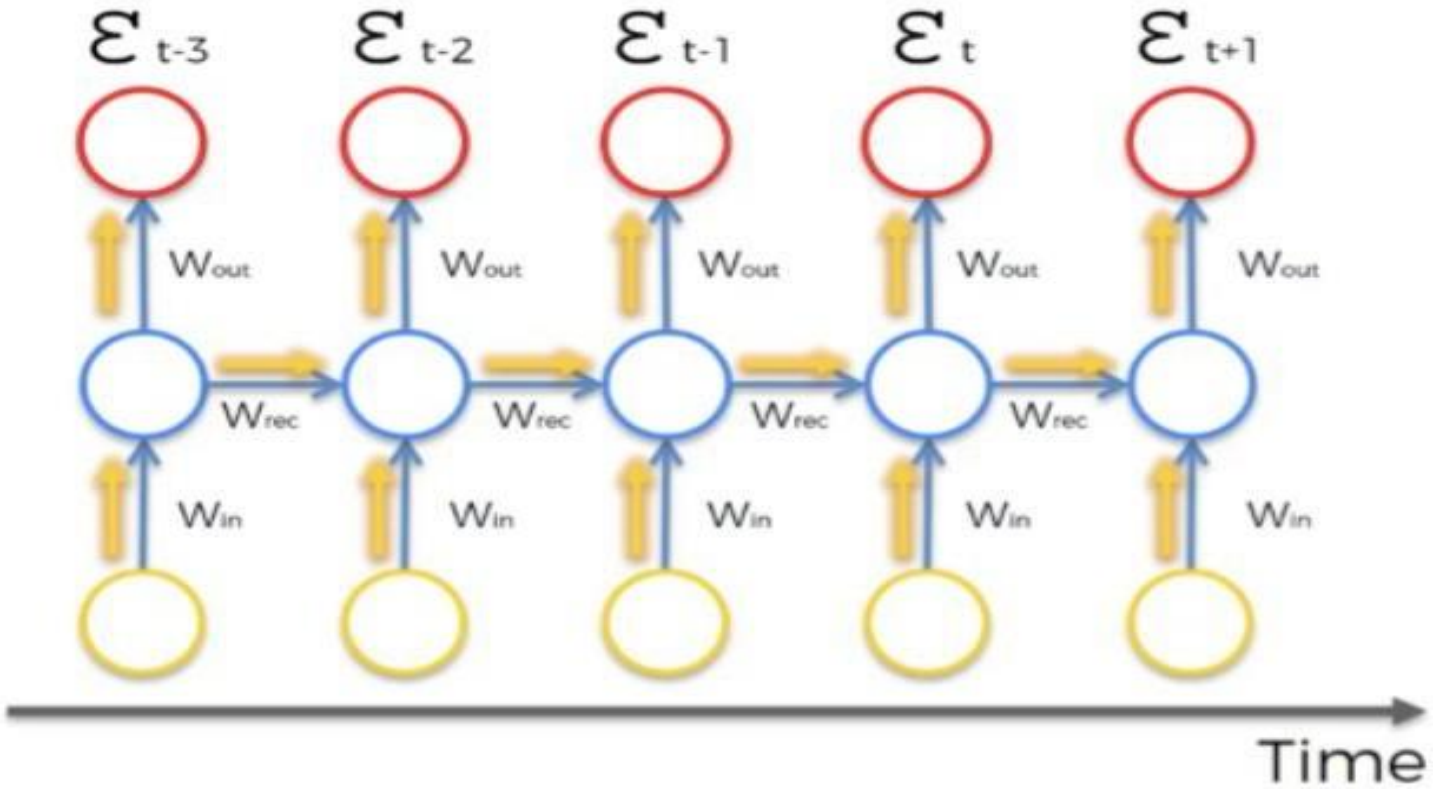


Limitations of BPTT

- BPTT has difficulty with local optima. Local optima are a more significant issue with recurrent neural networks than feed-forward neural networks.
- When using BPTT in RNN, we face problems such as exploding gradient and vanishing gradient.

Vanishing Gradient Problem

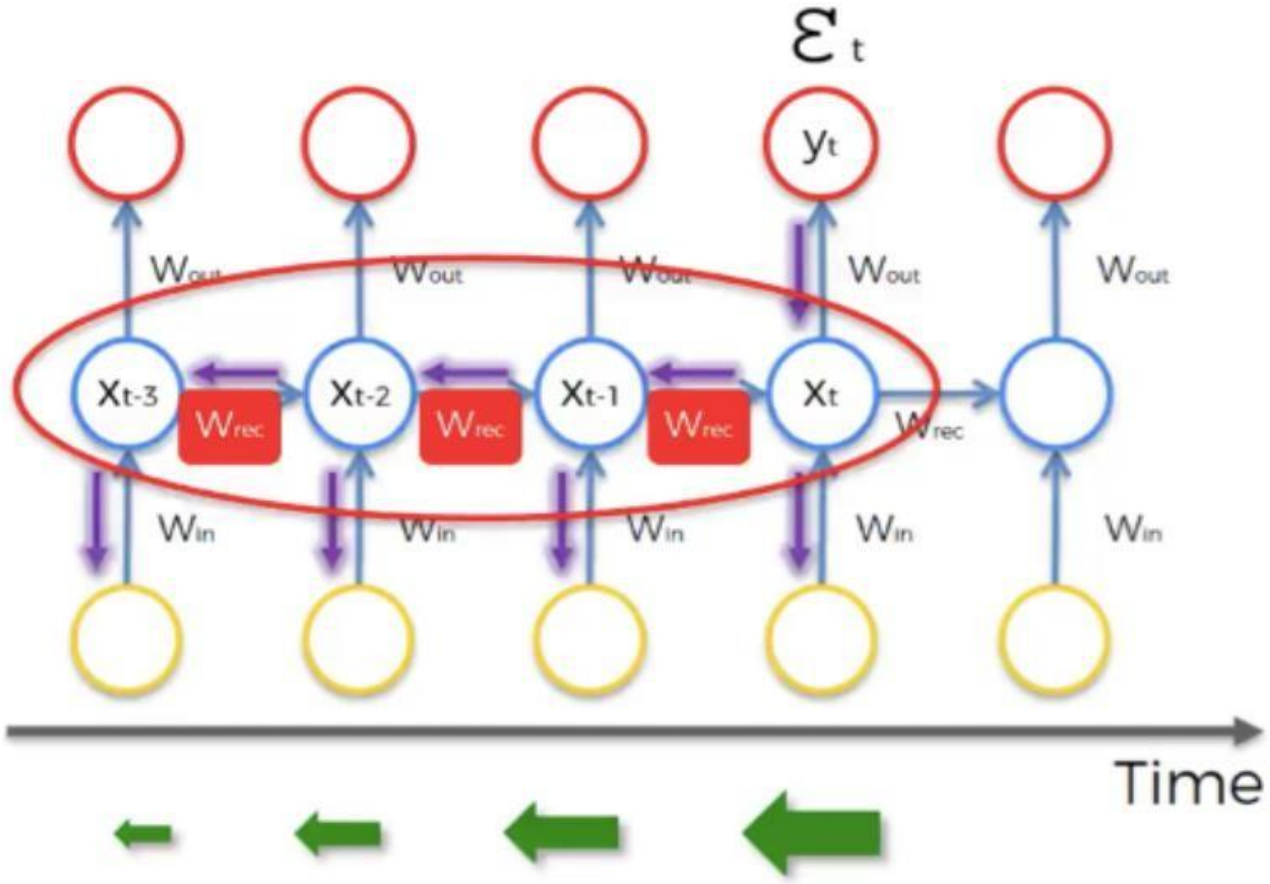
- During the training, your cost function compares your outcomes (red circles on the image below) to your desired output.
- As a result, you have these values throughout the time series, for every single one of these red circles.



- The problem relates to updating w_{rec} (weight recurring) – the weight that is used to connect the hidden layers to themselves in the unrolled temporal loop.

Vanishing Gradient Problem

- During the training, your cost function compares your outcomes (red circles on the image below) to your desired output.



$$\frac{\partial \mathcal{E}}{\partial \theta} = \sum_{1 \le t \le T} \frac{\partial \mathcal{E}_t}{\partial \theta} \quad (3)$$

$$\frac{\partial \mathcal{E}_t}{\partial \theta} = \sum_{1 \le k \le t} \left(\frac{\partial \mathcal{E}_t}{\partial \mathbf{x}_t} \frac{\partial \mathbf{x}_t}{\partial \mathbf{x}_k} \frac{\partial^+ \mathbf{x}_k}{\partial \theta} \right) \quad (4)$$

$$\frac{\partial \mathbf{x}_t}{\partial \mathbf{x}_k} = \prod_{t \ge i > k} \frac{\partial \mathbf{x}_i}{\partial \mathbf{x}_{i-1}} = \prod_{t \ge i > k} \mathbf{W}_{rec}^T diag(\sigma'(\mathbf{x}_{i-1})) \quad (5)$$

W_{rec} ~ small ⇨ Vanishing

W_{rec} ~ large ⇨ Exploding

Solutions to the Vanishing and Exploding Gradient Problem

In case of exploding gradient, you can:

- Stop backpropagating after a certain point, which is usually not optimal because not all of the weights get updated;
- Penalize or artificially reduce gradient;
- Put a maximum limit on a gradient.

In case of vanishing gradient, you can:

- Initialize weights so that the potential for vanishing gradient is minimized;
- Have Echo State Networks that are designed to solve the vanishing gradient problem;
- Have Long Short-Term Memory Networks (LSTMs).

Bidirectional RNN

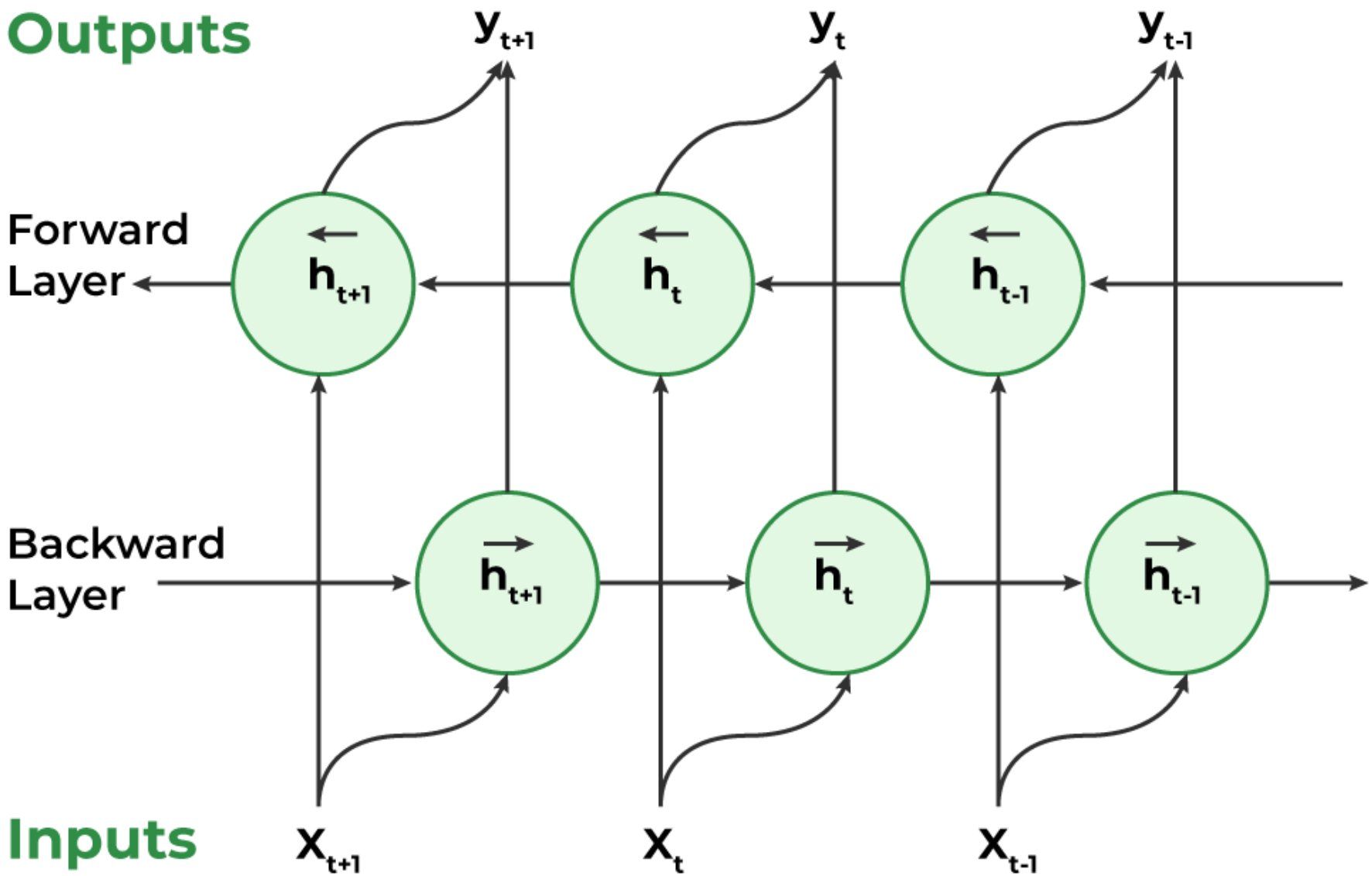
- A Bidirectional Recurrent Neural Network (Bi-RNN) is an advanced type of RNN that processes input sequences in both forward and backward directions to capture context from both past and future at every time step, which enables the network to utilize both past and future context when making predictions.

- Traditional RNNs suffer from the vanishing gradient problem where gradients become too small during backpropagation making training difficult. To address this, we have advanced RNN architectures like Bidirectional Recurrent Neural Network.

- It works like normal RNN by moving in forward direction, updating the hidden state depending on the current input and the prior hidden state at each time step.

- The backward hidden layer on the other hand analyses the input sequence in the opposite manner, updating the hidden state based on the current input and the hidden state of the next time step.

- **For example:** I like apple. It is very healthy.


Outputs
y_{t+1}
y_t
y_{t-1}
Forward Layer
$\overleftarrow{h}_{t+1}$
$\overleftarrow{h}_t$
$\overleftarrow{h}_{t-1}$
Backward Layer
$\overrightarrow{h}_{t+1}$
$\overrightarrow{h}_t$
$\overrightarrow{h}_{t-1}$
Inputs
x_{t+1}
x_t
x_{t-1}

Working of Bidirectional Recurrent Neural Network

1. **Inputting a Sequence:** A sequence of data points each represented as a vector with the same dimensionality is fed into a BRNN. The sequence might have different lengths.
2. **Dual Processing:** Both the forward and backward directions are used to process the data. The hidden state at time step **t** is determined in:
   - **Forward direction:** Based on the input at step **t** and the hidden state at step **t-1**.
   - **Backward direction:** Based on the input at step **t** and the hidden state at step **t+1**.

3. **Computing the Hidden State:** A non-linear activation function is applied to the weighted sum of the input and the previous hidden state. This creates a memory mechanism that enables the network to retain information from earlier steps.
4. **Determining the Output:** A non-linear activation function is applied to the weighted sum of the hidden state and output weights to compute the output at each step. This output can either be:
   - The final output of the network.
   - An input to another layer for further processing.

Advantages of BRNNs

- **Enhanced context understanding:** Uses both past and future data for better predictions.
- **Improved accuracy:** Useful in NLP and speech processing tasks.
- **Better handling of variable-length sequences:** More flexible compared to standard RNNs.
- **Increased robustness:** Can mitigate noise and irrelevant information due to forward and backward processing.

Challenges of BRNNs

- **High computational cost:** Requires twice processing compared to unidirectional RNNs.
- **Longer training time:** More parameters to optimize making convergence slow.
- **Limited real-time applicability:** Since they require entire sequence before making predictions they are not ideal for real-time applications like live speech recognition.
- **Less interpretability:** Understanding why a prediction was made is more complex than a standard RNN.

LSTM

Long Short Term Memory - Working principle of all gates in forward propagation, Intro about LSTM BPTT, Types of LSTM, Limitations - Bidirectional LSTMs

- LSTMs are a special type of Recurrent Neural Networks (RNNs) designed by **Hochreiter & Schmidhuber** **to handle long-term dependencies.**
- Traditional **RNNs struggle with vanishing and exploding gradients**, making it difficult for them to remember information over long sequences. **LSTMs solve this by introducing gates** that control the flow of information.
- **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.

Problem with Long-Term Dependencies in RNN

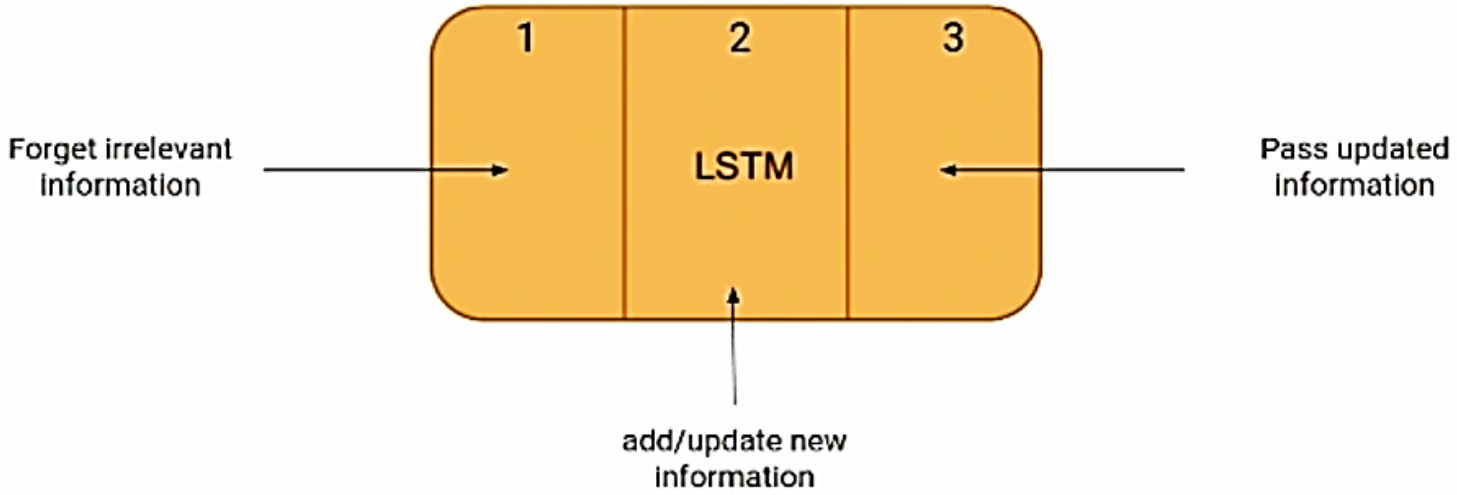
- **Vanishing Gradient:** When training a model over time, the gradients (which help the model learn) can shrink as they pass through many steps. This makes it hard for the model to learn long-term patterns since earlier information becomes almost irrelevant.

- **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.

LSTM Architecture

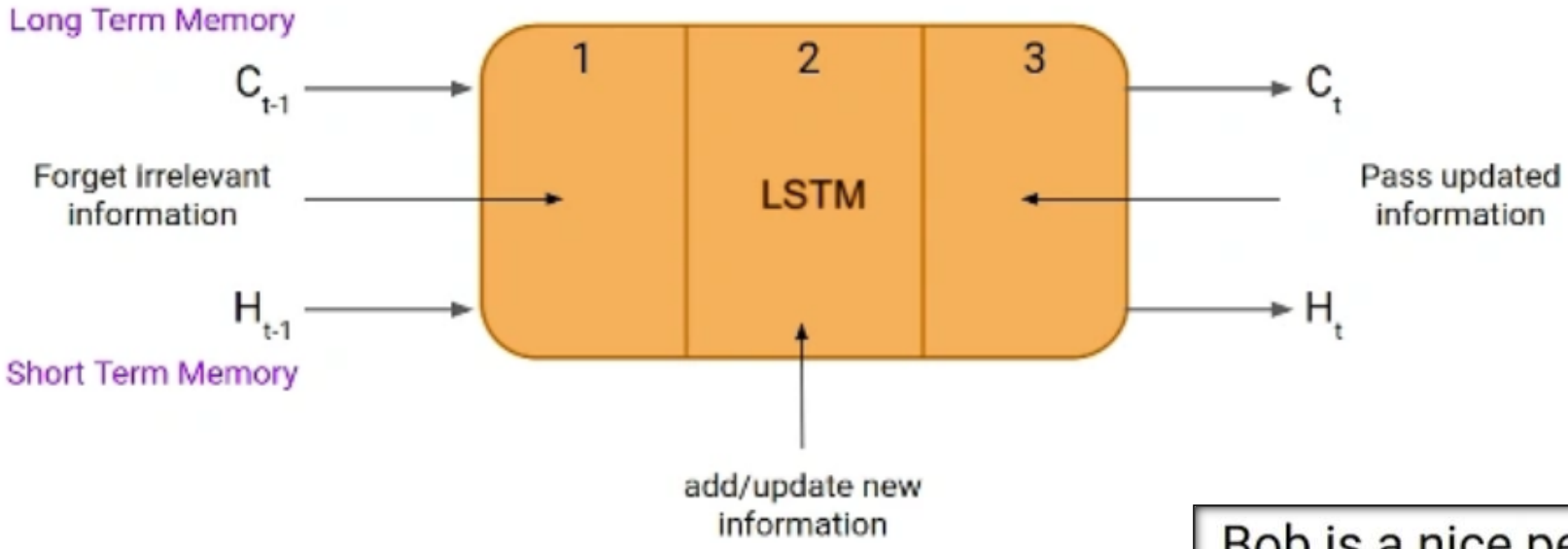
LSTM architectures involves the memory cell which is controlled by three gates: the input gate, the forget gate and the output gate. These gates decide what information to add to, remove from and output from the memory cell.

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



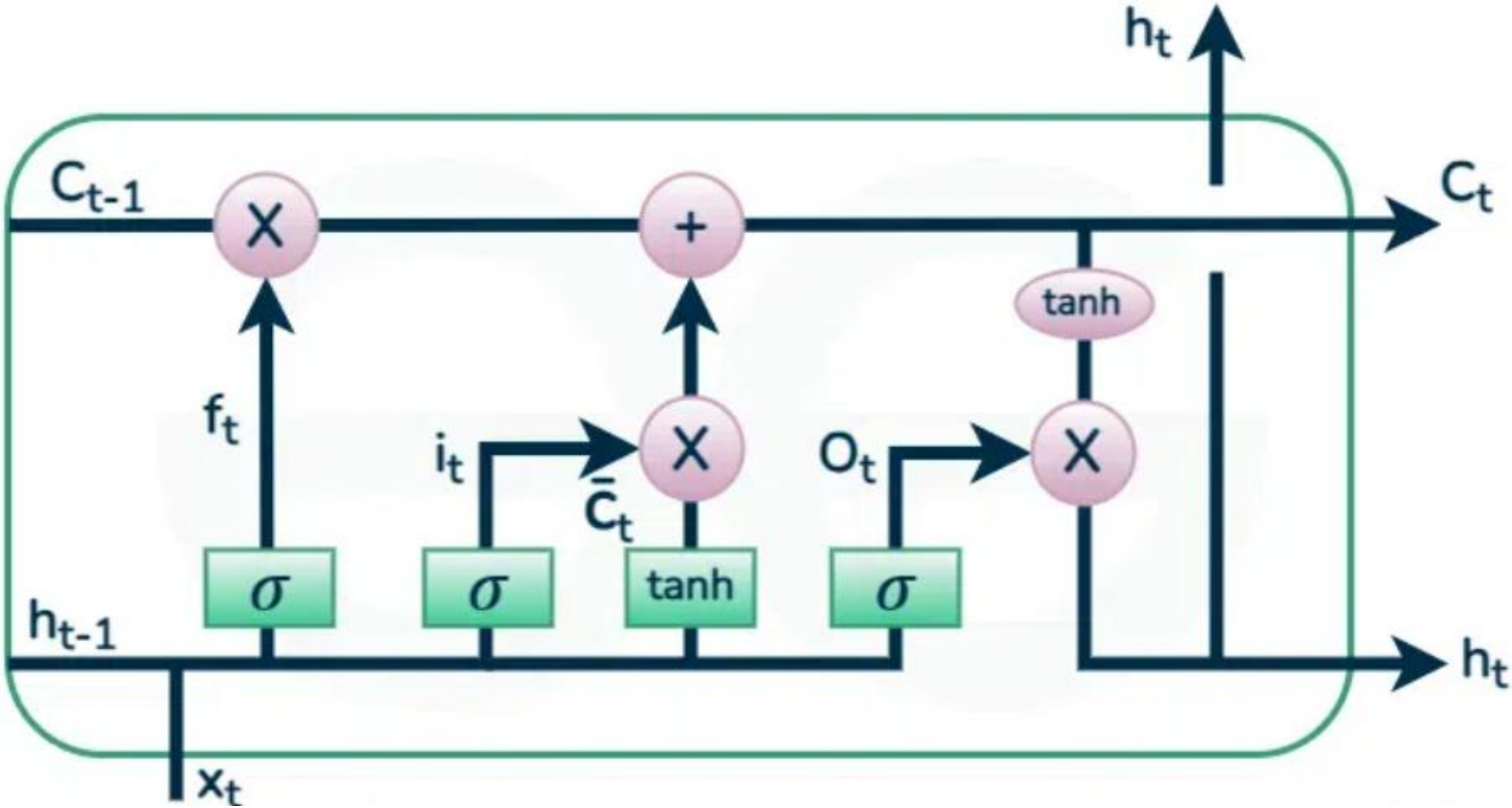
- The first part chooses whether the information coming from the previous timestamp is to be remembered or is irrelevant and can be forgotten.
- In the second part, the cell tries to learn new information from the input to this cell.
- At last, in the third part, the cell passes the updated information from the current timestamp to the next timestamp.
- This one cycle of LSTM is considered a single-time step.
- These three parts of an LSTM unit are known as gates. They control the flow of information in and out of the memory cell or lstm cell.

- The hidden state is known as Short term memory, and the cell state is known as Long term memory.
- Note that the cell state carries the information along with all the timestamps.



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

As we move from the first sentence to the second sentence, our network should realize that we are no more talking about Bob. Now our subject is Dan. Here, the Forget gate of the network allows it to forget about it.


h_t
C_{t-1}
X
+
C_t
tanh
f_t
i_t
X
O_t
X
$\bar{C}_t$
σ
σ
tanh
σ
h_{t-1}
h_t
x_t

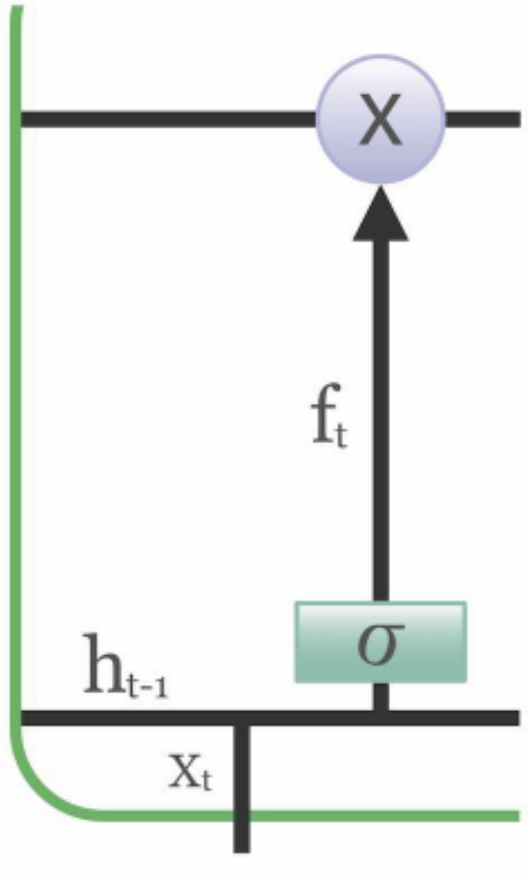
Three Gates in LSTM

1 Forget Gate (f_t) – "What should I forget?"

- **Purpose:** Controls which parts of the **cell state** (C_t) should be removed.
- **Formula:**

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

- W_f = Weights for the forget gate
- $[h_{t-1}, x_t]$ = Concatenation of previous hidden state and current input
- b_f = Bias term
- σ = Sigmoid activation (output is between 0 and 1)



- **How it Works:**
  - If f_t is **1**, the information is **kept**.
  - If f_t is **0**, the information is **forgotten**.

- **Example:**
  - **Sentence 1:** "Bob is a nice person." (Stores information about Bob)
  - **Sentence 2:** "Dan, on the other hand, is evil."
    - **Forget gate** reduces Bob's importance, making space for Dan.
  - **Sentence 3:** "Bob helps people a lot."
    - **Forget gate removes Dan's details**, allowing Bob's characteristics to return.

2 Input Gate (i_t) – "What new information should I store?"

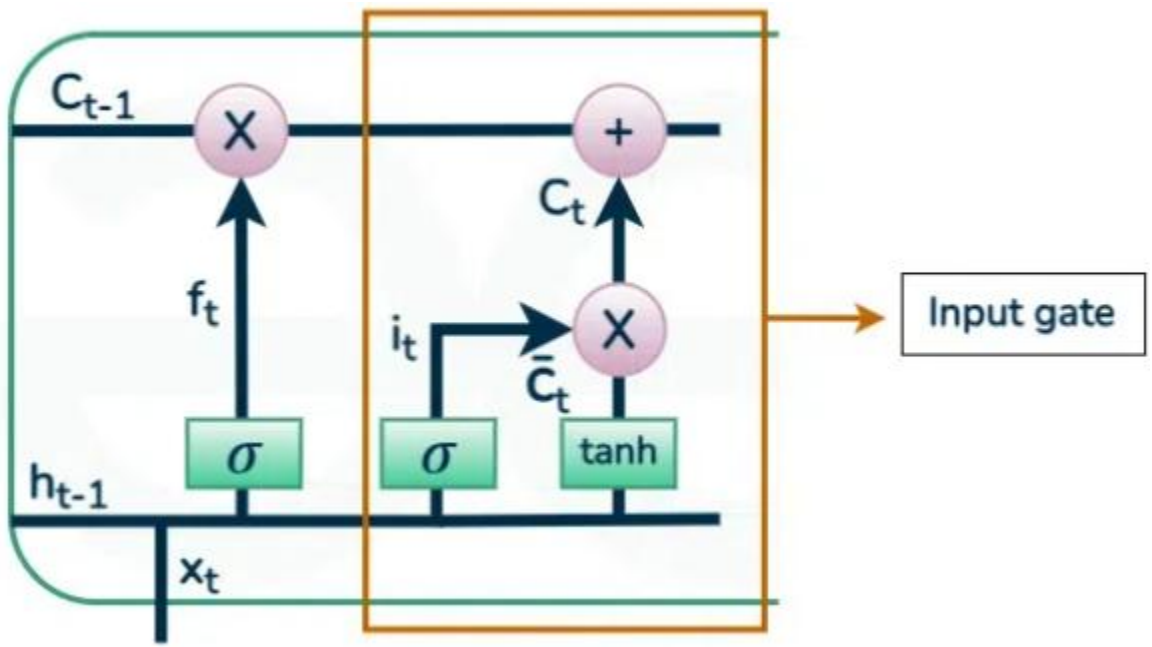
- **Purpose:** Controls what **new information** should be added to the **cell state** (C_t).
- **Formula:**

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

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

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

- W_i, W_C = Weights for the input and candidate cell state
- $\tilde{C}_t$ = Candidate cell state, created using **tanh** activation
- C_t = Updated cell state



- How it Works:
  - i_t decides **how much of the new information ($\tilde{C}_t$) should be added to** C_t.

- Example:
- **Sentence 2:** "Dan, on the other hand, is evil."
  - **Forget gate removes Bob.**
  - **Input gate adds information about Dan being evil.**
- **Sentence 3:** "Bob helps people a lot."
  - **Forget gate removes Dan.**
  - **Input gate updates Bob's information with 'helping people'.**

3 Output Gate (o_t) – "What should be the next hidden state?"

- **Purpose:** Decides what **part of the cell state** should be **sent to the next step** as the hidden state (h_t).
- **Formula:**

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

$$h_t = o_t \cdot \tanh(C_t)$$

- W_o = Weights for the output gate
- h_t = Hidden state (short-term memory)

- **How it Works:**

- The **output gate selects useful information** from the cell state to pass as output.

- **Example:**

- If we're predicting the **next word** in a sentence, the output gate determines what's **most relevant right now**.
- In **Sentence 3**, since Bob is back in focus, the output gate ensures the hidden state **reflects Bob's updated details**.

Step	Forget Gate (f_t)	Input Gate (i_t)	Output Gate (o_t)
Sentence 1: "Bob is nice."	Removes old data	Stores info about Bob	Sends Bob's info to next step
Sentence 2: "Dan is evil."	Removes Bob's details	Adds Dan's traits	Sends Dan's details
Sentence 3: "Bob helps people."	Removes Dan's details	Reintroduces Bob	Updates hidden state with Bob

LSTM BPTT

- One of the fundamental techniques used to train LSTMs is Backpropagation Through Time (BPTT) where we have sequential data.
- In **feedforward neural networks** this process is simple because the data moves in one direction from input to output.
- However in RNNs and LSTMs the situation is a bit different as they process sequential data (like sentences, time-series or speech) meaning that each piece of data depends on previous pieces. This creates a **temporal dependency** between the steps. Because of this we can't just apply backpropagation normally. Instead we need to modify it so that it can account for **time dimension,** this is where **Backpropagation Through Time (BPTT)** comes in.

HOW BPTT WORKS IN LSTM

1. **Forward pass**: Compute the outputs (h_t, C_t).
2. **Compute the loss**: Compare predicted and actual outputs.
3. **Backward pass**:
    - Compute the gradients of the loss with respect to LSTM parameters using the **chain rule**.
    - Since LSTMs use **sigmoid and tanh**, they mitigate the **vanishing gradient** problem seen in standard RNNs.
4. **Update weights**: Using an optimization algorithm (e.g., Adam, SGD).

- x_t be the input at timestep t.
- h_{t-1} and h_t be the hidden states at previous and current timestep.
- c_{t-1} and c_t be the cell states at previous and current timestep.
- σ represents the **sigmoid activation function**.
- anhrepresents the **hyperbolic tangent activation function**.
- $\odot$ represents element-wise (Hadamard) multiplication.

The initial values of c_0 and h_0 are usually set to **zero**.

Step 1: Initialization of Weights

Each LSTM cell has three gates: Input Gate, Forget Gate and Output Gate each with associated weight matrices and biases.

Input Gate:

- **Weights:** W_{xi}, W_{hi}, b_i
- **Candidate Cell State Weights:** W_{xg}, W_{hg}, b_g

Forget Gate:

- **Weights:** W_{xf}, W_{hf}, b_f

Output Gate:

- **Weights:** W_{xo}, W_{ho}, b_o

Step 2: Forward Pass Through LSTM Gates

At each timestep t we compute the gate activations as follows:

- **Forget Gate:** $f_t = \sigma(W_{xf}x_t + W_{hf}h_{t-1} + b_f)$
- **Input Gate:** $f_t = \sigma(W_{xf}x_t + W_{hf}h_{t-1} + b_f)$
- **Candidate Cell State:** $g_t = \tanh(W_{xg}x_t + W_{hg}h_{t-1} + b_g)$
- **Cell State Update:** $c_t = f_t \odot c_{t-1} + i_t \odot g_t$
- **Output Gate:** $o_t = \sigma(W_{xo}x_t + W_{ho}h_{t-1} + b_o)$
- **Hidden State Update:** $h_t = o_t \odot \tanh(c_t)$

Step 3: Backpropagation Through Time (BPTT) in LSTMs

1. Gradient of Loss w.r.t Output Gate

$$\frac{dE}{do_t} = \frac{dE}{dh_t} \odot \tanh(c_t)$$

2. Gradient of Loss w.r.t Cell State

$$\frac{dE}{dc_t} = \frac{dE}{dh_t} \odot o_t \odot (1 - \tanh^2(c_t))$$

3. Gradient of Loss w.r.t Input Gate and Candidate Cell State

$$\frac{dE}{di_t} = \frac{dE}{dc_t} \odot g_t$$

$$\frac{dE}{dg_t} = \frac{dE}{dc_t} \odot i_t$$

4. Gradient of Loss w.r.t Forget Gate

$$\frac{dE}{df_t} = \frac{dE}{dc_t} \odot c_{t-1}$$

5. Gradient of Loss w.r.t Previous Cell State

$$\frac{dE}{dc_{t-1}} = \frac{dE}{dc_t} \odot f_t$$

The goal is to compute gradients for all weights using the **chain rule** considering LSTM-specific gates and memory cells.

Step 4: Compute Weight Gradients

Using chain rule we compute gradients for weights associated with each gate.

Gradients for Output Gate Weights

$$\frac{dE}{dW_{xo}} = \frac{dE}{do_t} \odot o_t(1-o_t) \odot x_t$$

$$\frac{dE}{dW_{ho}} = \frac{dE}{do_t} \odot o_t(1-o_t) \odot h_{t-1}$$

$$\frac{dE}{db_o} = \frac{dE}{do_t} \odot o_t(1-o_t)$$

Using these gradients we update the LSTM weights during training.

Backpropagation Through Time (BPTT) for LSTMs involves computing **partial derivatives for each gate** propagating gradients backward over multiple timestamps and **updating weights using gradient descent**.

ADVANTAGES OF LSTM

1. **Solves the Vanishing Gradient Problem** – LSTMs use gates to control information flow, allowing them to retain information over long sequences.
2. **Handles Long-Term Dependencies Well** – The cell state acts as long-term memory, helping recall information across long gaps.
3. **Effective for Sequence Data** – Works well for speech recognition, text generation, and time-series forecasting.
4. **Flexible with Variable-Length Input** – Can process sequences of different lengths, making it useful for NLP tasks like machine translation.
5. **Works with Noisy Data** – The forget gate helps remove irrelevant details, making LSTMs robust to noisy or missing data.

DISADVANTAGES OF LSTM

1. **Computationally Expensive** – Requires more parameters, leading to higher memory consumption and slower training.
2. **Prone to Overfitting** – Can easily memorize training data, requiring regularization techniques like dropout.
3. **Difficult to Tune** – Needs careful selection of hyperparameters such as learning rate, batch size, and number of hidden units.
4. **Struggles with Very Long Sequences** – Still faces challenges in handling extremely long dependencies, where transformers perform better.
5. **Slower Inference Time** – Cannot process input in parallel, making it slower than transformer-based models like BERT or GPT.

BIDIRECTIONAL LSTM (BILSTM)

- A Bidirectional LSTM (BiLSTM) is an extension of the standard LSTM that processes input sequences **in both forward and backward directions**. This helps the model capture **past and future context** more effectively, making it useful for NLP tasks, speech recognition, and more.
- Standard LSTM processes input from left to right (past to future). BiLSTM consists of two LSTMs:
  - **Forward LSTM** → Reads the sequence from start to end.
  - **Backward LSTM** → Reads the sequence from end to start.
- The outputs of the two LSTM networks are then combined to produce the final output.
- LSTM networks can be stacked to form deeper models allowing them to learn more complex patterns in data. Each layer in the stack captures different levels of information and time-based relationships in the input.

For each time step t:

1. **Forward LSTM computes:**

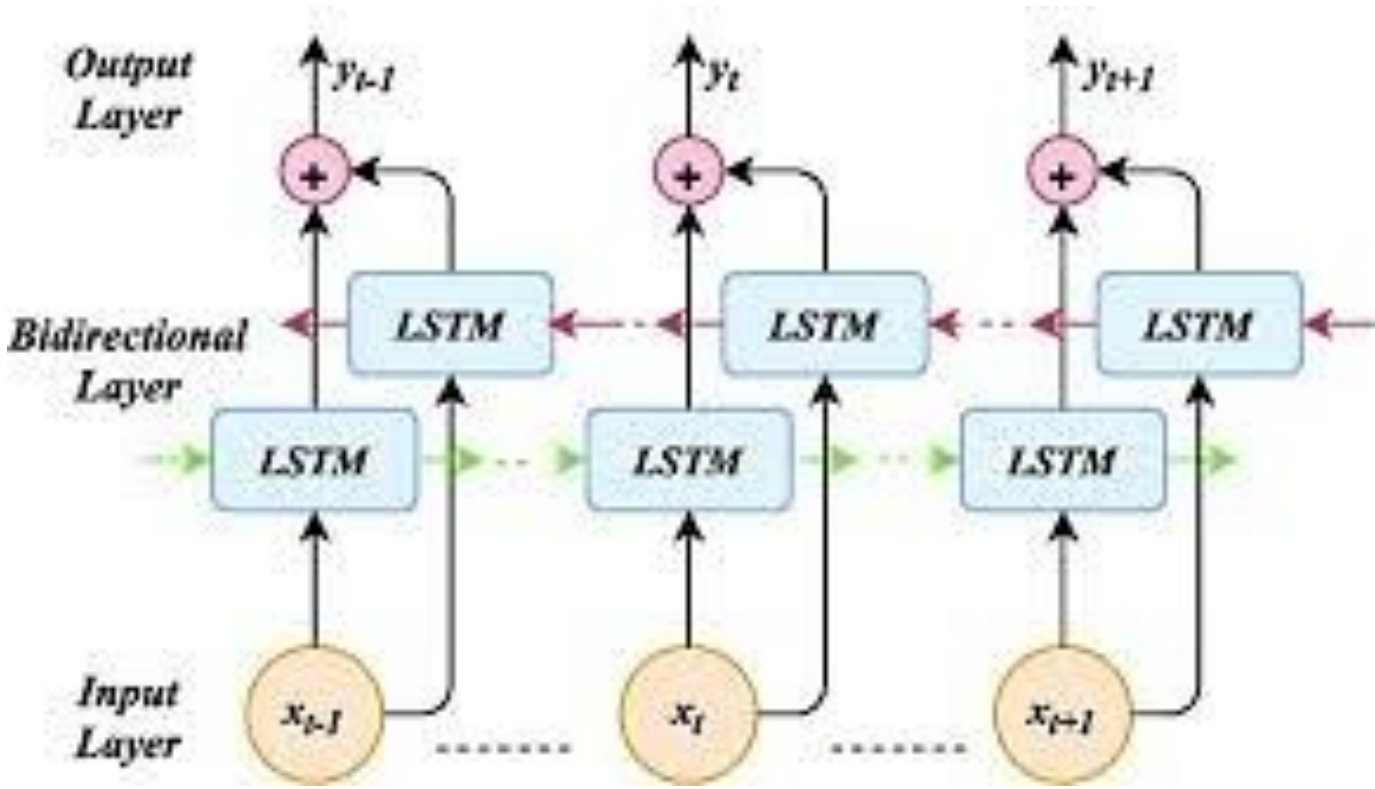
$$\overrightarrow{h_t} = LSTM_f(x_t, \overrightarrow{h_{t-1}})$$

2. **Backward LSTM computes:**

$$\overleftarrow{h_t} = LSTM_b(x_t, \overleftarrow{h_{t+1}})$$

3. **Final Output** is:

$$h_t = \text{Concat}(\overrightarrow{h_t}, \overleftarrow{h_t})$$



Advantages of BiLSTM

1. **Captures Past and Future Context** – Unlike regular LSTMs, BiLSTMs process sequences in both directions, providing **more context** for each word.
2. **Improved Performance in NLP Tasks** – Helps in **POS tagging, machine translation, text classification, and speech recognition**.
3. **Better Handling of Dependencies** – Useful when future words help determine the meaning of a previous word.

Disadvantages of BiLSTM

1. **Higher Computational Cost** – Uses **twice the number of parameters** compared to standard LSTM.
2. **Not Suitable for Real-Time Applications** – Requires **entire sequence** before processing, making it unsuitable for real-time systems.

TYPES OF LSTM

LSTM Classic (Standard LSTM)

- The **basic LSTM** model that processes input **from past to future** (left to right).
- Uses **forget, input, and output gates** to manage memory and handle long-term dependencies.
- **Example:** Sentiment analysis, text classification, time-series prediction.

Stacked LSTM (Deep LSTM)

- Has **multiple LSTM layers stacked** on top of each other.
- Helps the model **learn more complex patterns** in data.
- Requires **more computational power** but improves performance.
- **Example:** Handwriting recognition, deep speech processing.

Bidirectional LSTM (BiLSTM)

- Uses **two LSTMs**:
  - **Forward LSTM** → Reads the sequence from start to end.
  - **Backward LSTM** → Reads the sequence from end to start.
- Helps capture **both past and future context** in a sentence.
- **Example:** Named Entity Recognition (NER), machine translation, speech recognition.

Gated Recurrent Unit (GRU)

- A **simplified version of LSTM** with only two gates: **update gate and reset gate**.
- **Faster and less computationally expensive** than LSTM.
- Performs well in many tasks **without losing accuracy**.
- **Example:** Chatbots, stock price prediction, real-time applications.

Bidirectional GRU (BGRU)

- Similar to **BiLSTM** but uses **GRU instead of LSTM**.
- Processes data **in both forward and backward directions** for better context understanding.
- **Lighter and faster** than BiLSTM while maintaining good accuracy.
- **Example:** Speech-to-text conversion, text translation.

ConvLSTM (Convolutional LSTM)

- Combines **CNN (Convolutional Neural Network) with LSTM**.
- Used for **spatial and temporal data** (e.g., images/videos changing over time).
- Helps LSTM understand **visual sequences**.
- **Example:** Video classification, weather forecasting, medical image analysis.

LSTMs With Attention Mechanism

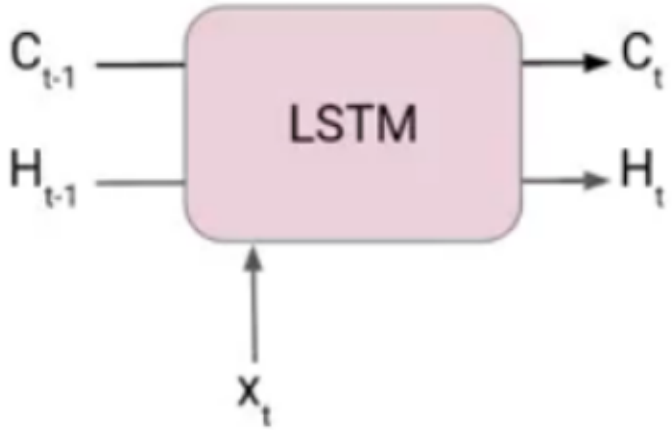
- Uses an **attention mechanism** to focus on the most important parts of the input sequence.
- Helps LSTM **pay more attention** to relevant words or frames instead of treating all equally.
- Improves **translation, summarization, and question-answering** tasks.
- **Example:** Google Translate, text summarization, chatbot responses.

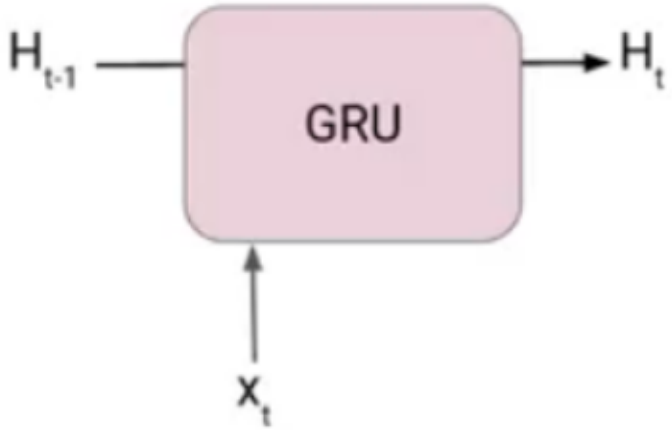
LSTM Type	Key Feature	Use Case
LSTM Classic	Standard memory-based LSTM	Sentiment analysis, time-series prediction
BiLSTM	Processes input in both directions	Speech recognition, NER, machine translation
Stacked LSTM	Multiple LSTM layers for deeper learning	Handwriting recognition, speech processing
GRU	Simplified LSTM with fewer gates	Chatbots, stock price prediction
BGRU	BiLSTM alternative with GRU	Text translation, speech-to-text
ConvLSTM	LSTM combined with CNN for spatial & temporal data	Video classification, weather forecasting
LSTMs With Attention	Focuses on important inputs for better performance	Machine translation, summarization

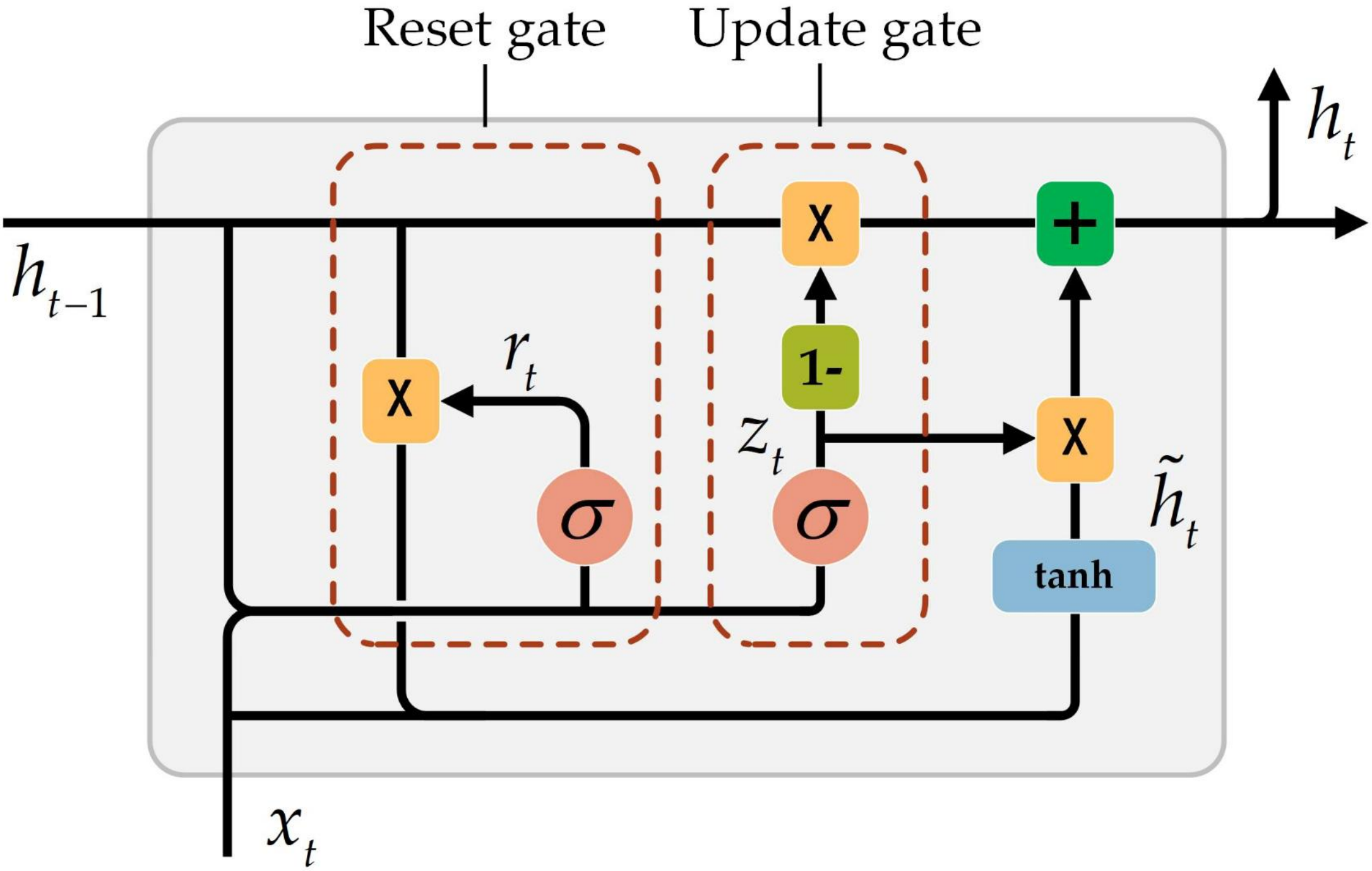
GATED RECURRENT UNITS

Gated Recurrent Units – Working principle of all gates in forward propagation, Limitations

- Gated Recurrent Units (GRUs) are a type of RNN introduced by Cho et al. in 2014.
- The core idea behind GRUs is to use gating mechanisms to selectively update the hidden state at each time step allowing them to remember important information while discarding irrelevant details.
- GRUs aim to simplify the LSTM architecture by merging some of its components and focusing on just two main gates: the update gate and the reset gate.
- Just like LSTM, GRU uses gates to control the flow of information. Unlike LSTM, GRU does not have a separate cell state (Ct). It only has a hidden state(Ht). Due to the simpler architecture, GRUs are faster to train.






Reset gate
Update gate
h_t
h_{t-1}
X
+
r_t
X
1-
z_t
X
$\tilde{h}_t$
σ
σ
tanh
x_t

Step 1: Compute Reset Gate

$$r_t = \sigma(W_r x_t + U_r h_{t-1} + b_r)$$

- W_r, U_r are weights, b_r is the bias.
- σ is the sigmoid activation function.
- r_t determines how much of the past hidden state should be forgotten.

Step 2: Compute Update Gate

$$z_t = \sigma(W_z x_t + U_z h_{t-1} + b_z)$$

- W_z, U_z are weights, b_z is the bias.
- z_t decides how much of the previous hidden state is retained.

Step 3: Compute Candidate Hidden State

$$\tilde{h}_t = \tanh(W_h x_t + U_h(r_t \odot h_{t-1}) + b_h)$$

- $r_t \odot h_{t-1}$ means the reset gate controls how much of the past hidden state is used.

Step 4: Compute Final Hidden State

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t$$

- The update gate z_t controls whether the new candidate state $\tilde{h}_t$ should replace the previous hidden state h_{t-1}.

Advantages of GRU

- **Faster Training and Efficiency:** Compared to LSTMs (Long Short-Term Memory networks), GRUs have a simpler architecture with fewer parameters. This makes them faster to train and computationally less expensive.
- **Effective for Sequential Tasks:** GRUs excel at handling long-term dependencies in sequential data like language or time series. Their gating mechanisms allow them to selectively remember or forget information, leading to better performance on tasks like machine translation or forecasting.
- **Less Prone to Gradient Problems:** The gating mechanisms in GRUs help mitigate the vanishing/exploding gradient problems that plague standard RNNs. This allows for more stable training and better learning in long sequences.

Disadvantages of GRU

- **Less Powerful Gating Mechanism:** While effective, GRUs have a simpler gating mechanism compared to LSTMs which utilize three gates. This can limit their ability to capture very complex relationships or long-term dependencies in certain scenarios.
- **Potential for Overfitting:** With a simpler architecture, LSTM and GRU Architecture might be more susceptible to overfitting, especially on smaller datasets. Careful hyperparameter tuning is crucial to avoid this issue.
- **Limited Interpretability:** Understanding how a GRU Activation Function arrives at its predictions can be challenging due to the complexity of the gating mechanisms. This makes it difficult to analyze or explain the network's decision-making process.

Feature	LSTM (Long Short-Term Memory)	GRU (Gated Recurrent Unit)
Gates	3 (Input, Forget, Output)	2 (Update, Reset)
Cell State	Yes it has cell state	No (Hidden state only)
Training Speed	Slower due to complexity	Faster due to simpler architecture
Computational Load	Higher due to more gates and parameters	Lower due to fewer gates and parameters
Performance	Often better in tasks requiring long-term memory	Performs similarly in many tasks with less complexity