UNIT-5

RECURRENT NUERAL NETWORKS

Introducing sequential data, Modelling sequential data - order matters, Representing sequences, RNN for modelling sequences, Understanding the RNN, Hidden ~~occur~~ recurrence. The challenges of learning long-~~term~~ range interactions. Long short term memory

Ref. Pg. No. 567 - 583. Prescribed Text2

Introducing Sequential data

Let us begin our discussion of RNNs by looking at the nature of sequential data, which is more commonly known as Sequences or Sequence data.

Modelling Sequential data

Typical machine learning algorithms for Supervised learning assume that the training examples are mutually independent and have the same underlying distribution. In this regard, based on mutual independance assumption, the order in which ~~we~~

the training examples are given to the model is irrelevant. An example of this scenario would be the Iris dataset that we previously worked with.

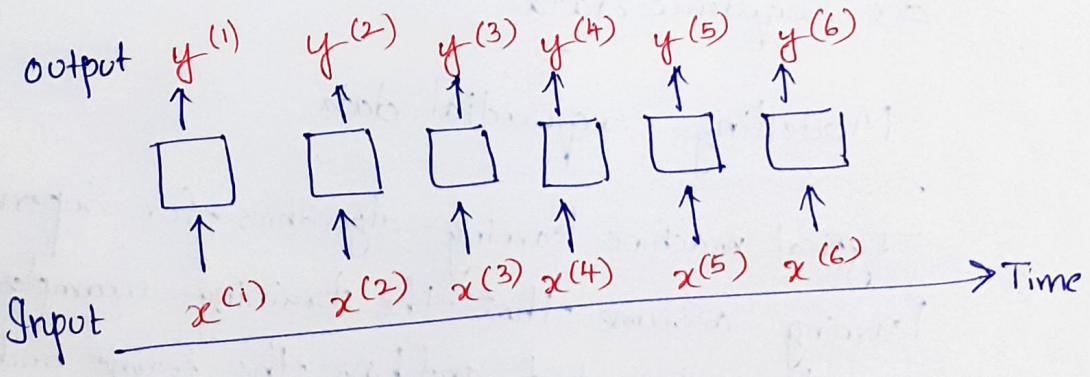
However, this assumption is not valid when we deal with sequences - by definition, order matters. Predicting the market value of a particular stock would be an example of this scenario.

Representing sequences

We have established that order among data points is important in sequential data,

Let $x^{(1)}, x^{(2)} \ldots x^{(T)}$ be the instances in order of the sequences and the length of the sequences is T.

Let $x^{(t)}$ be the instance at a particular time, t.

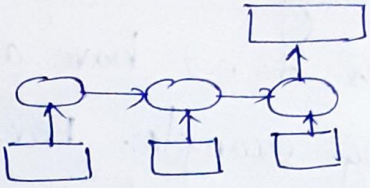


The Neural Network Models such as MLP- Multi layer perceptron and CNNs, assume that the training examples are independent of each other and thus do not incorporate ordering information. We can say that such models do not have a memory of previously seen training examples. For instance, the samples are passed through the feedforward and backpropagation steps, and the weights are updated independently of the order in which the training examples are processed.

RNNs (Recurrent Neural Networks) are designed for modelling sequences and are capable of remembering past information and processing new events accordingly, which is a clear advantage when working with sequence data.

The different categories of sequence modeling

Many-to-one

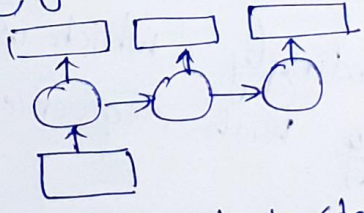


The input data is a sequence, but the output is a fixed size vector or scalar, not a sequence.

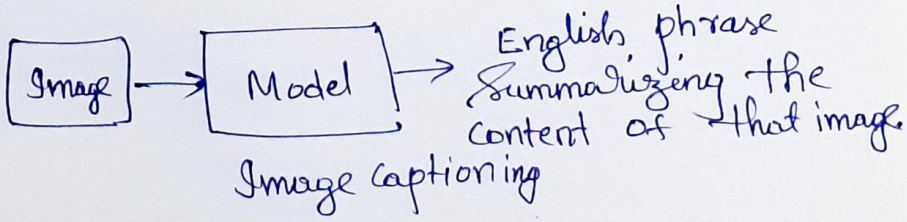
For example, in sentiment analysis, the input is a text based and output is a class label

Ex:

One to Many



The input data is in standard format and not a sequence, but the output is a sequence



Many-to-Many

Both the input and output arrays are sequences.

This category can be further divided based on whether the input and output are synchronized.

Ex: (1) A synchronized many to many modeling task is Video classification, where each frame in a Video is labelled.

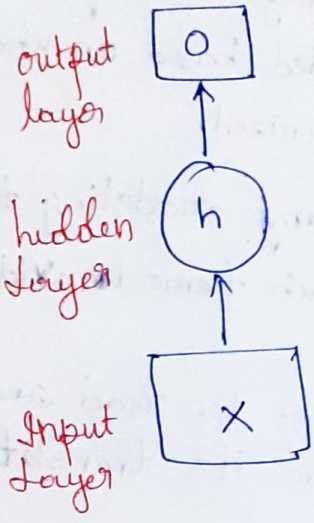
(2) Entire English Sentence must be read and processed by a machine before its translation into German is produced.

RNNs for modeling Sequences

In this Section, before we start implementing RNNs in TensorFlow, we will discuss the main concepts of RNNs. We will begin by looking at the typical structure of an RNN, which includes a recursive component to model sequence data.

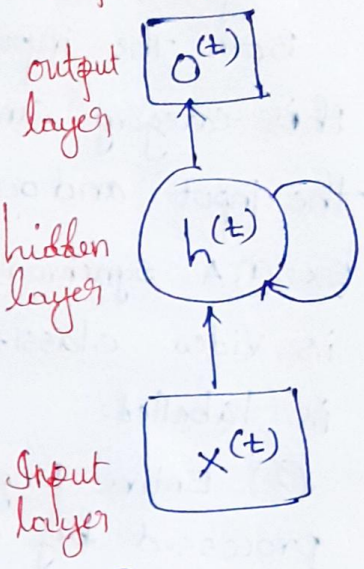
Then, we will examine how the neuron activations are computed in a typical RNNs and we will then discuss Solutions to these challenges, such as LSTM and gated recurrent units (GRUs)

Understanding the RNN looping mechanism



A standard feedforward network

output layer are vectors



Recurrent neural network

Determining the type of output from an RNN

This generic RNN architecture could correspond to the two sequence modeling categories where the input is a sequence.. Typically, a recurrent layer can return a sequence as output, $< O^{(0)}, O^{(1)}, \ldots, O^{(T)} >$ or simply return the last output (at $t=T$, that is $O^{(T)}$).

If it is many to many $\Rightarrow$ $< O^{(0)}, O^{(1)} \ldots O^{(T)} >$

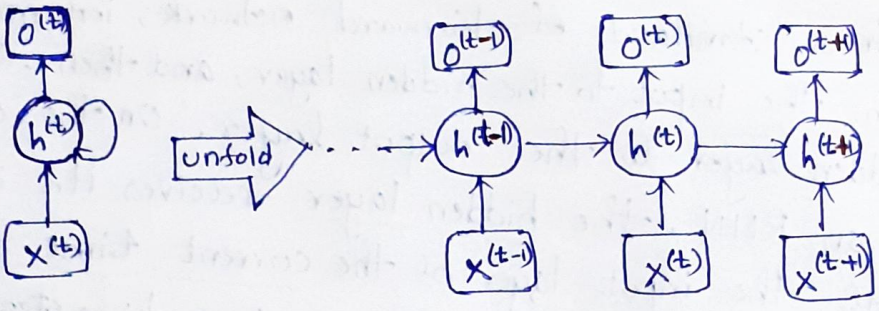
many to one $\Rightarrow$ $O^{(T)}$

*) In a standard feedforward network, information flows from the input to the hidden layer, and then from the hidden layer to the output layer. On the other hand, in an RNN, the hidden layer receives its input from both the input layer of the current time step and the hidden layer from the previous time step.

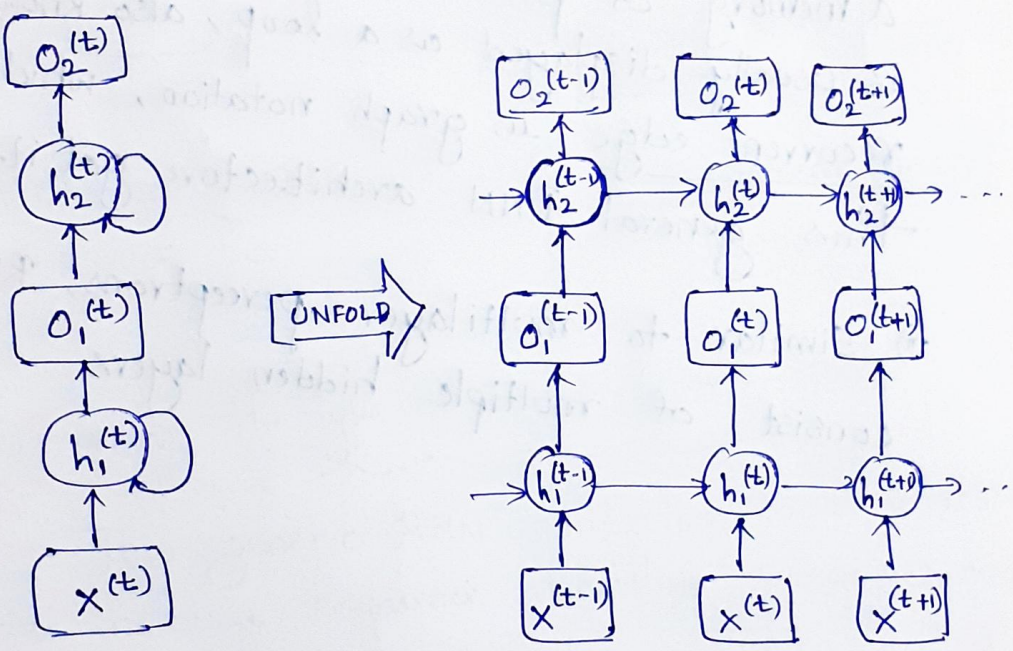
*) The flow of information in adjacent time steps in the hidden layer allows the network to have a memory of past events. This flow of information is usually displayed as a loop, also known as a recurrent edge in graph notation, which is how this general RNN architecture got its name.

*) Similar to multilayer perceptrons, RNNs can consist of multiple hidden layers.

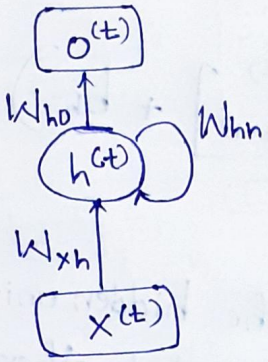
Single layer RNN



Multilayer RNN



Computing activations in an RNN



$$W_h = [W_{hh} : W_{xh}] \rightarrow (1)$$

Let W_{xh}: The weight matrix between the input, $x^{(t)}$, and the hidden layer, h

W_{hh}: The weight matrix associated with recurrent edge

W_{ho}: The weight matrix between the hidden layer and output layer

In certain implementations, you may observe that the weight matrices, W_{xh} and W_{hh}, are concatenated to a combined matrix, $W_h = [W_{xh}; W_{hh}]$

$$Z_h^{(t)} = W_{xh} x^{(t)} + W_{hh} h^{(t-1)} + b_h$$

$$h^{(t)} = \phi_h(Z_h^{(t)}) = \phi_h[W_{xh} x^{(t)} + W_{hh} h^{(t-1)} + b_h]$$

Where b_h = bias vector for hidden units

$\phi_h(\cdot)$ = Activation function for the hidden units

In case you may want to use the concatenated weight matrix, $W_h = [W_{xh} : W_{hh}]$, the formula for computing hidden units will change, as follows

$$h^{(t)} = \phi_h\left([W_{xh}; W_{hh}]\begin{bmatrix} x^{(t)} \\ h^{(t-1)} \end{bmatrix} + b_h\right)$$

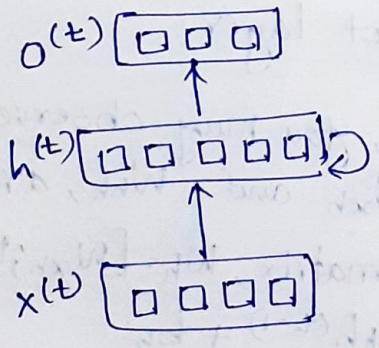
(ϕ_h: Activation function)

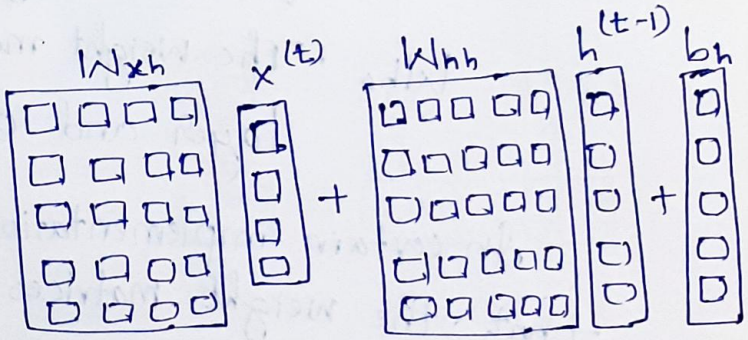
Once the activations of the hidden units at the current time step are computed, then the activations of the output units will be computed as follows,

$$o^{(t)} = \phi_o\left(W_{ho} h^{(t)} + b_o\right)$$

(ϕ_o: o/p, W_{ho}: o/p)

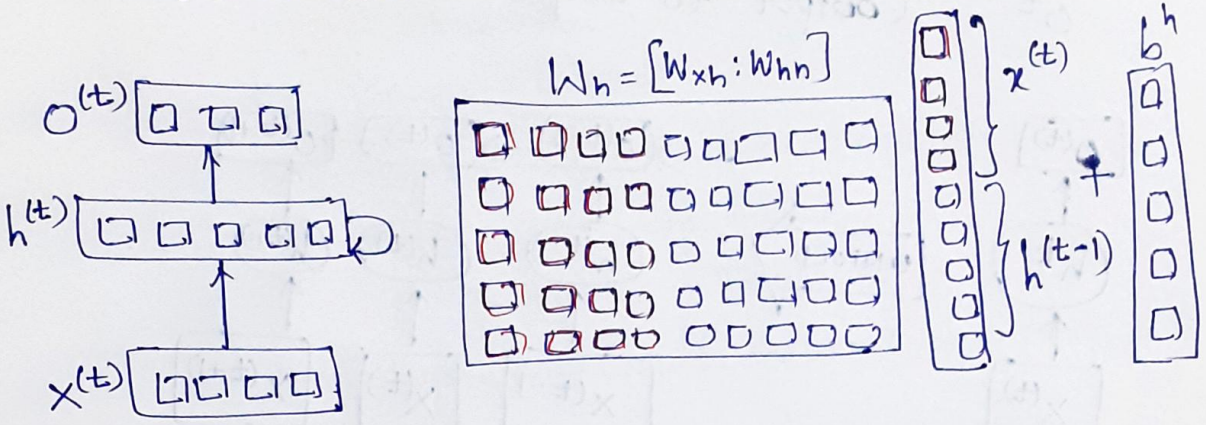
Formulation 1





$$o^{(t)} = \phi_o\left[W_{ho} h^{(t)} + b_o\right] = \text{Final output}$$

Formulation 2

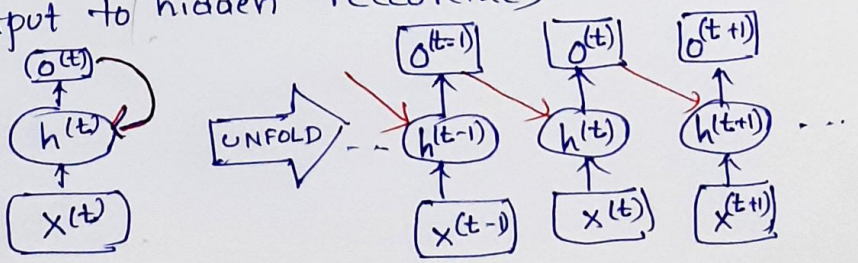


$$O^{(t)} = \phi_o (W_{ho} h^{(t)} + b_o) = \text{Final output}$$

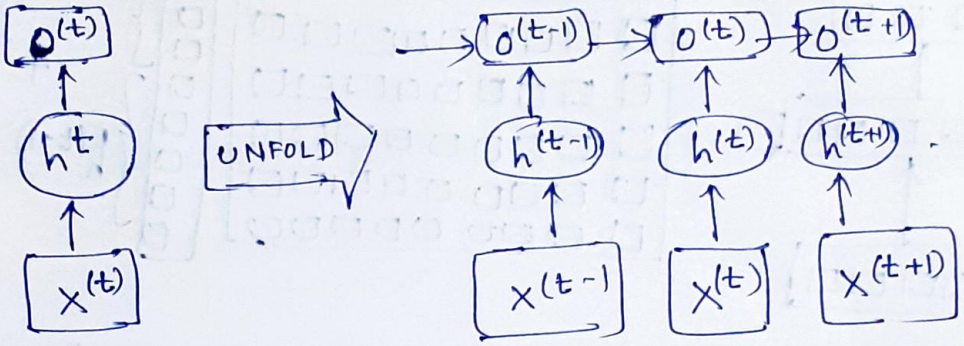
Hidden-recurrence versus output-recurrence

So far, you have seen recurrent networks in which the hidden layer has the recurrent property. However, note that there is an alternative model in which recurrent connection comes from the output layer. In this case, the net activations from the output layer at the previous time stem, O^{t-1}, can be added in one of two ways:

(1) To the hidden layer at the current time step, h^t (output to hidden reccurence)



(2) To the output layer at the current time step, o^t .. (output to output recurrence)



---

(1) To illustrate this consider the following program which uses output to output recurrence.

Let W_{hh} = hidden to hidden recurrence weight matrix

W_{oh} = output to hidden recurrence weight matrix

W_{oo} = output to output recurrence weight matrix

Program to Implement Simple RNN

(1)

```
# Program to implement Simple RNN
import tensorflow as tf
tf.random.set_seed(1)
rnn_layer = tf.keras.layer.SimpleRNN(
    units=2, use_bias=True,
    return_sequences = True)
rnn_layer.build(input_shape= (None, None, 5))
W_xh, W_oo, b_h = rnn_layer.weights
print('W_xh shape: ', W_xh.shape)
print('W_oo shape: ', W_oo.shape)
print('b_h  shape: ', b_h.shape)
x_seq = tf.convert_to_tensor(
    [[1.0]*5, [2.0]*5, [3.0]*5],
    dtype = tf.float32)
## output of SimpleRNN
output = rnn_layer(tf.reshape(x_seq,
                   shape = (1, 3, 5)))
```

.... Contd. in next page

```
## manually computing the output
out_man = []
for t in range(len(x_seq)):
    xt = tf.reshape(x_seq[t], (1,5))
    print('Time step {} => ', format(t))
    print('  Input          : ', xt.numpy())
    ht = tf.matmul(xt, w_xh) + b_h
    print('  Hidden         : ', ht.numpy())
    if t>0:
      prev_o = out_man[t-1]
    else:
        prev_o = tf.zeros(shape=(ht.shape))
    ot = ht + tf.matmul(prev_o, w_oo)
    ot = tf.math.tanh(ot)
    out_man.append(ot)
    print('  output (manual) : ', ot.numpy())
    print('  SimpleRNN output: ', format(t),
                  output[0][t].numpy())

    print()
```

Program-1 output

Program-1 : To implement RNN

```
W_xh shape : (5,2)
W_oo shape: (2,2)
b_h  shape: (2,)
```

The weights of input to hidden layer : W_xh

```
1 [[-0.62200572  0.74339894]
2  [0.24251717 -0.12119704]
3  [-0.38525409  0.2638626]
4  [0.8809386  -0.12017238]
5  [0.2964511  -0.19422936]]
```

The weights of the output to output layer : W_oo

```
1 [[0.98796964  0.15464693]
1  [-0.15464693  0.98796984]]
```

The weights of bias of hidden layer : b_h

```
[0, 0]
Input
tf.tensor
[[1, 1, 1, 1, 1]
 [2, 2, 2, 2, 2]
 [3, 3, 3, 3, 3]] shape (3,5), dtype=float32
```

continued in next page

It is for t=0

Time step 0 ⇒

Input xt : [[1. 1. 1. 1.]]

It is w_xh : [[-0.6200572 0.7433989]
[0.242517 -0.12119704]
[-0.38525409 0.2638626]
[0.8809836 -0.12017238]
[0.2964511 0.19422936]]

It is b_h : [0 0]

Hidden computation

ht = xt * w_xh + b_h : [[0.41464037 0.96012145]]

It is inside else for t=0

It is ht [[0.41464037 0.96012145]]

It is prev_o = [[0. 0.]]

It is w_oo = [[0.98796964 0.15464693] [From RNN
[-0.15464693 0.9879698]]

It is ht = [0.41464037 0.96012145]]

It is ot before tanh

It is ot = ht + prev_o * w_oo → before tanh = [0.41464037, 0.96012145]

tanh(0.4146) tanh(0.9601)

output (manual) : [[0.39240566 0.74433]]

Simple RNN output : [0.39240566 0.74433]

It is for t=1

Time step 1 ⇒

Input xt = : [2 2 2 2 2]

It is w_xh : [[-0.6200572 0.7433989]
[0.242517 -0.12119704]
[-0.38525409 0.2638626]
[0.8809836 -0.12017238]
[0.2964511 0.1942936]]

It is b_h : [0 0]

ht = xt * w_xh + b_h : [[0.8292807 3 1.9202429]]

It is prev_o : [[0.39240566 0.74433106]]

It is w_oo : [[0.98796964 0.15464693]
[-0.15464693 0.9879698]]

It is ht : [[0.82928073 1.9202429]]

It is ot = ht + prev_o * w_oo before tanh

[[1.1018571 2.7163038]]

SimpleRNN output : [[0.80116504 0.991247]] tanh(1.1018571) tanh(2.7163)

output (manual) : [[0.80116504 0.991247]]

SimpleRNN output : [0.80116504 0.991247]

It is for t = 2

Time step2 ⇒

Input xt = : [[3. 3. 3. 3. 3.]]

It is w_xh : [[-0.6200572 0.7433989]
[0.242517 -0.1211]
[-0.38525409 0.2638]
[0.8809836 -0.1201]
[0.2964 0.1942]]

It is b_h : [0. 0.]

ht = xt * w_xh + b_h : [[1.2438 2.8803]]

It is prev_o [[0.80116504 0.991247]]

It is w_oo [[0.98796964 0.15464693]
[-0.15464693 0.9879698]]

It is ht [[1.2438 2.8803]]

It is ot = ht + prev_o * w_oo before tanh

[[1.8821471 3.9836311]] tanh(1.8821) tanh(3.98363)

output (manual) : [[0.95468 0.9993069]]

SimpleRNN output : [0.95468 0.9993069]

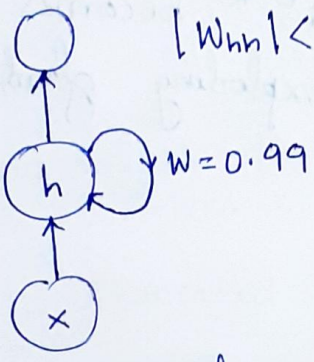
The challenges of learning long-range interactions

The BPTT introduces there will be

(i) Vanishing gradient problem

(ii) Exploding gradient problem

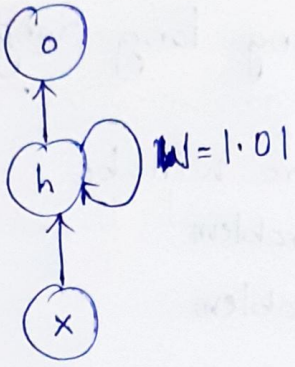
Vanishing gradient



This problem arises because

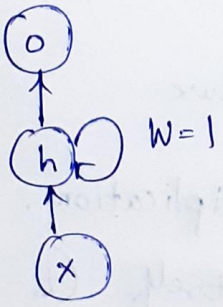
$\frac{dh^{(t)}}{dh^{(k)}}$ has $(t-k)$ multiplications. Therefore we have to multiply W by itself $(t-k)$ which results is W^{t-k} if $|W| < 1$, then this factor becomes very small when $(t-k)$ is large.

Exploding gradient



$\frac{dh^{(t)}}{dh^{(k)}}$ has $(t-k)$ multiplications of W. If $|W|>1$, then W^{t-k} becomes very large. This is known as exploding gradient problem

Desirable



$\frac{dh^{(t)}}{dh^{(k)}}$ has $(t-k)$ multiplication of W's i.e $W^{t-k}=1$ if $|W|=1$. It avoids vanishing gradient and exploding gradient problem.

In order to Solve this Vanishing gradient and exploding gradient, there are three Solutions namely

(i) Gradient clipping
(ii) TBPTT - Truncated Backpropagation through time
(iii) LSTM - Long Short Term Memory

(i) Gradient Clipping :- Using Gradient Clipping, we Specify a cut-off value or threshold value for the gradients. We assign this cut-off value to gradient values that exceeds the cut-off value. Using this it Solves exploding gradient problem.

(ii) TBPTT :- Truncated Backpropagation through time

TBPTT simply Solve the exploding gradient problem by limiting the number of steps that the gradient can effectively flow back and properly update the weights

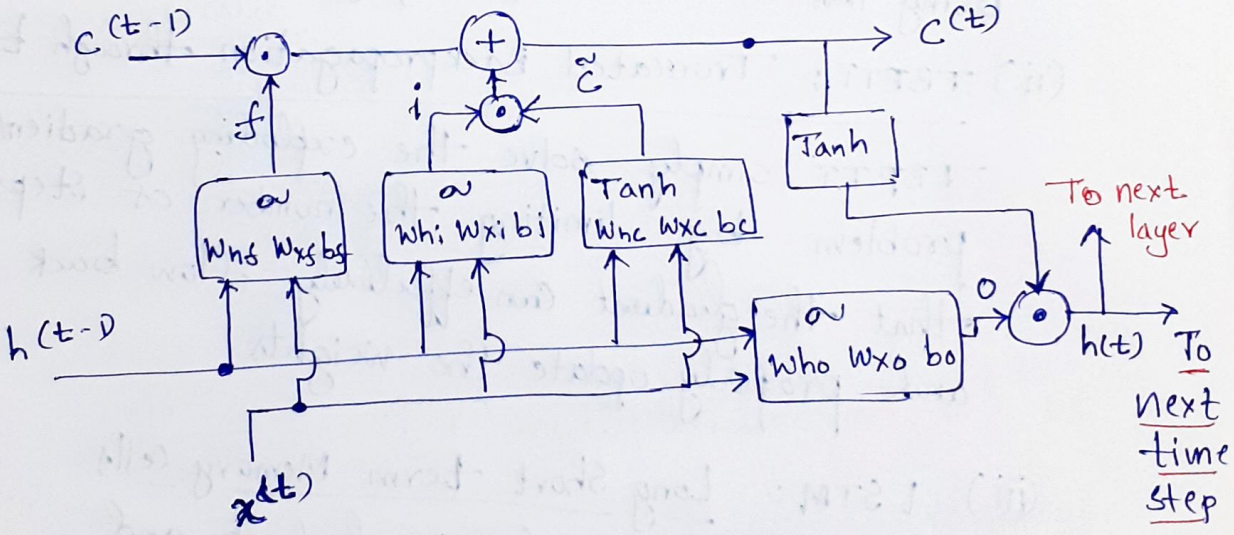
(iii) LSTM :- Long Short term Memory Cells

LSTM Solve exploding gradient and Vanishing gradient through the use of memory cells.

Long short-term Memory Cells

The basic building block of an LSTM cell is a memory cell, which essentially represents or replaces the hidden layer of standard RNNs.

In each memory cell, there is a recurrent edge that has the desirable weight $w=1$, to overcome the vanishing and exploding gradient problems. The unfolded structure of a modern LSTM cell is shown in the following figure



⊙ = Element-wise multiplication

⊕ = Element-wise summation

$x^{(t)}$ = Input data at time 't'

$h^{(t-1)}$ = hidden units at time "t-1"

σ = Sigmoid activation function

tanh = Tanh = Tanh activation function

(i) f_t = forget gate

$$f_t = \sigma\left(W_{xf} x^{(t)} + W_{hf} h^{(t-1)} + b_f\right)$$

forget gate (f_t) allows the memory cell to reset the cell state without growing indefinitely.

(ii) i_t = input gate

$$i_t = \sigma\left(W_{xi} x^{(t)} + W_{hi} h^{(t-1)} + b_i\right)$$

(iii) $\tilde{C}_t$ = candidate value

$$\tilde{C}_t = \tanh\left(W_{xc} x^{(t)} + W_{hc} h^{(t-1)} + b_c\right)$$

(iv) $$C^{(t)} = \left(C^{(t-1)} \odot f_t\right) \oplus \left(i \odot \tilde{C}_t\right)$$

Where $C^{(t)}$ = cell state at time 't'

(v) O_t = output gate

$$O_t = \sigma\left[W_{xo} x^{(t)} + W_{ho} h^{(t-1)} + b_o\right]$$

(vi) $h^{(t)}$ = hidden units

$$h^{(t)} = O_t \odot \tanh(c^t)$$

Recurrent edge : The flow of information in adjacent time steps in the hidden layer allows the network to have a memory of past events. This flow of information is displayed as a loop, also known as a recurrent edge in graph notation.