Lecture Notes

Recurrent Neural Networks

In this module, you learnt about the recurrent neural network (RNN) and its different types of architectures. You learnt about the different kinds of RNN architectures and their usage on problems that involve sequences. Then, you learnt about backpropagation and the problem of vanishing and exploding gradient in RNNs. You also learnt about bidirectional RNNs which are variants of the standard RNNs. Finally, you learnt about the LSTM network and its variant GRU network both of which help in tackling the problem of vanishing gradients faced by a vanilla RNN network.

Sequential Data

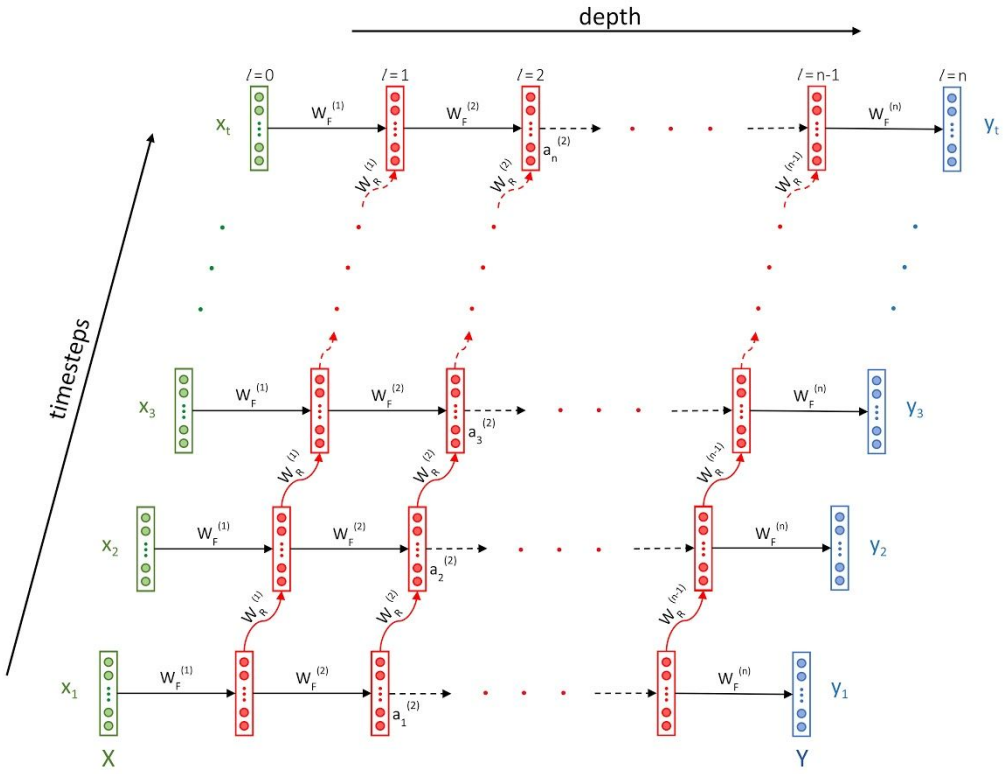
You learnt that a normal neural network is insufficient to train sequence data. Some examples of sequence data are:

- Time series
- Music
- Videos
- Text

You learnt that sequential data contains **multiple entities**. The **order** in which these entities are present is important.

Architecture of an RNN

You learnt about the **architecture** of an RNN. The architecture is such that it takes into account the multiple entities present in a sequence. The architecture of an RNN and the feedforward equations are shown below:



$$z_t^{(l)} = W_F^{(l)} a_t^{(l-1)} + W_R^{(l)} a_{t-1}^{(l)} + b^{(l)}$$

$$a_t^{(l)} = f^{(l)}(z_t^{(l)})$$

Figure 1: Architecture of an RNN

You learnt that an RNN consists of **recurrent layers**. The weights present below the recurrent layers are denoted by $\mathbf{W_R}$. You also learnt that W_R is a **square matrix** because it connects each and every neuron at timestamp 't' in layer 'l' with each and every neuron at timestep 't+1' in the same layer 'l'.

You also learnt that each activation is dependent on two things: the activation in the previous layer 'l-1' at the current timestep 't', and the activation in the same layer 'l' at the previous timestep 't-1'.

You also learnt about the matrix sizes of the terms involved in the feedforward equations. The following table shows the matrix sizes:

Table 1: Matrix sizes in an RNN network

Term	Size
$W_F^{(l)}$	(#neurons in layer l, #neurons in layer l-1)

$W_R^{(l)}$	(#neurons in layer l, #neurons in layer l)
$b^{(l)}$	(#neurons in layer l, 1)
$z_t^{(l)}$	(#neurons in layer l, batch_size)
$a_t^{(l)}$	(#neurons in layer l, batch_size)

In the above notation, 't' denotes the timestep, 'l' denotes the layer in the network and batch_size is the number of data points passed in one go.

You also learnt that tthere's one more way to write the feedforward equation in an RNN shown below:

$$z_t^{(l)} = W^{(l)}[a_t^{(l-1)}, a_{t-1}^{(l)}] + b^{(l)}$$

where,
$W^{(l)} = [\ W_F^{(l)} \mid W_R^{(l)}]$, that is the column-wise concatenation of the weight matrices at layer 'l'.
$[a_t^{(l-1)}, a_{t-1}^{(l)}]$ is the row-wise concatenation of the activations $a_t^{(l-1)}$ and $a_{t-1}^{(l)}$.

Types of RNN

Next, you went through the different types of RNN. You learnt that changing the input and/or the output leads to a different architecture. The different types of RNN that you learnt about are:

- **Many-to-one architecture**:

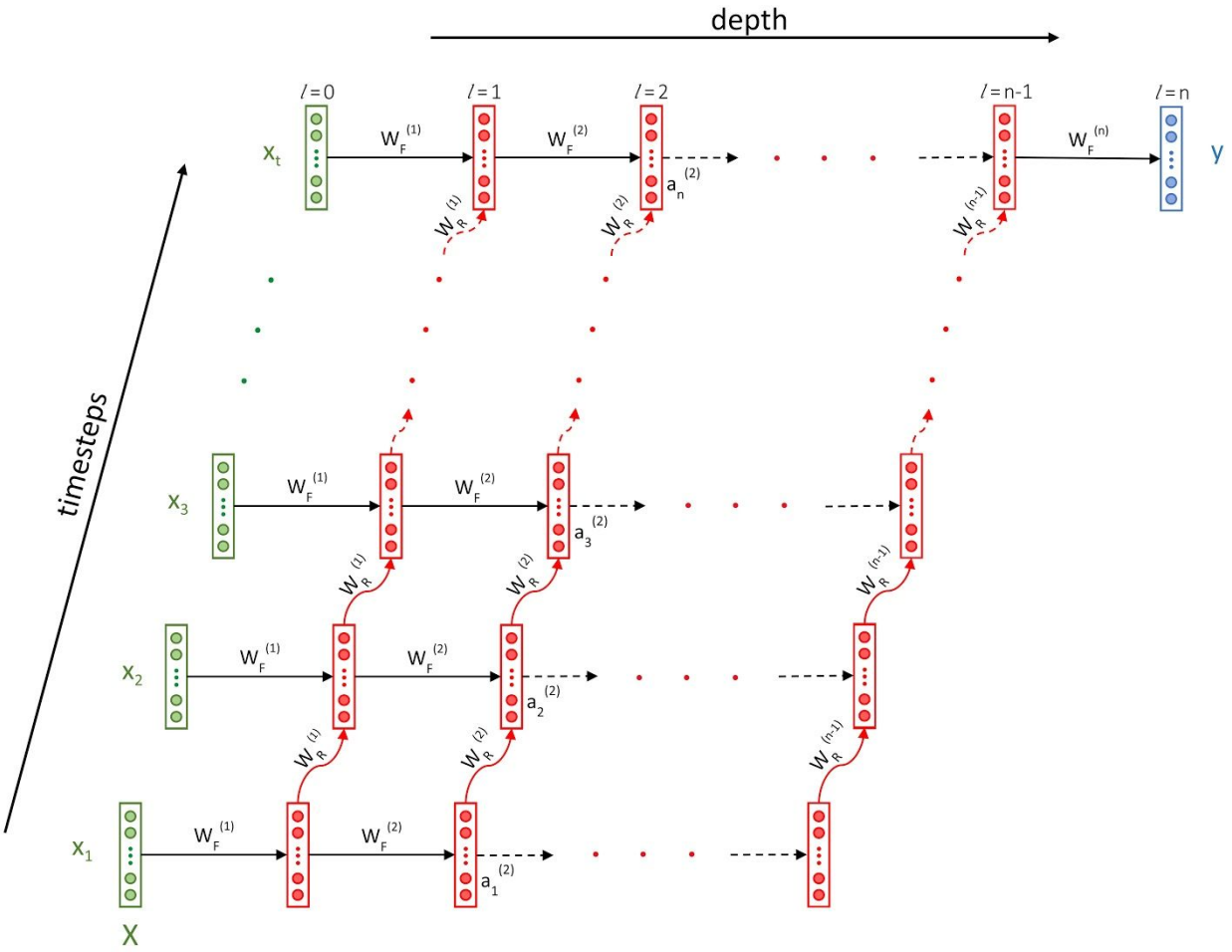


Figure 2: A many-to-one architecture of an RNN

You learnt that this architecture involves a **sequence as an input** and a **single entity as an output**. You used this architecture in the C-code generator which was a character level text generator.

- **Standard many-to-many architecture**:

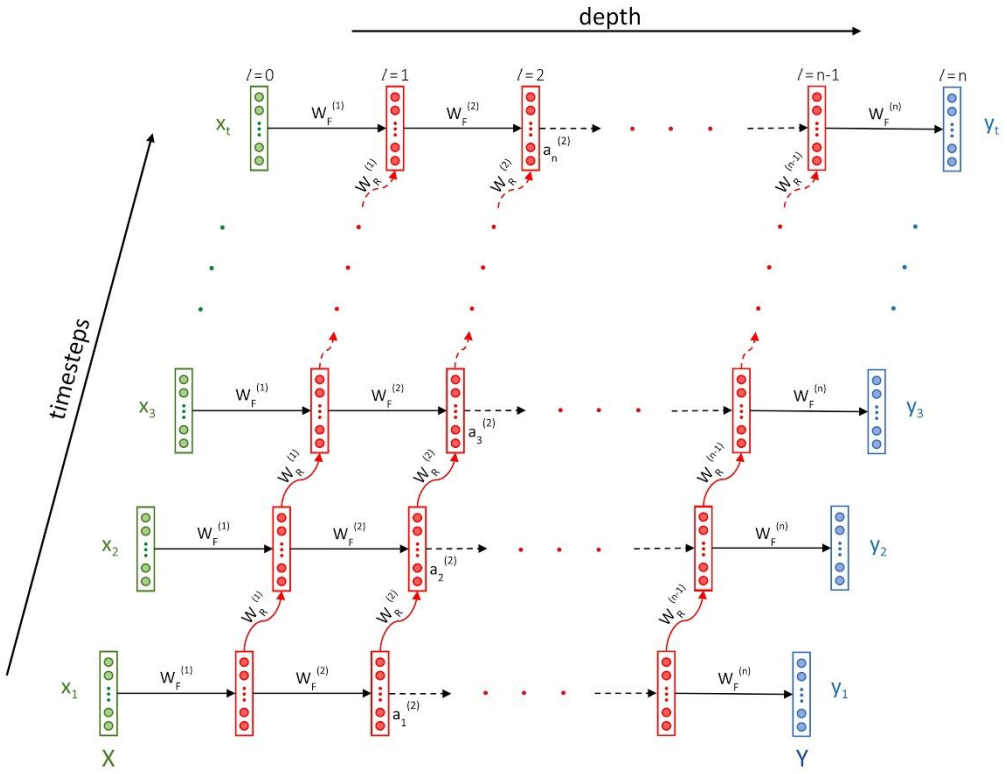


Figure 3: A standard many-to-many architecture of an RNN

You learnt that this type of RNN can be used to model data which involves **sequences in the input as well as the output**. The important thing to note here is that the input and output sequences must have a **one-to-one correspondence** and therefore the input and output sequences are **equal in length**. You used this type of architecture while building a POS tagger where the input comprised of a sentence and the output comprised of a part-of-speech tag for each word in the sentence.

- **Encoder-decoder architecture**:

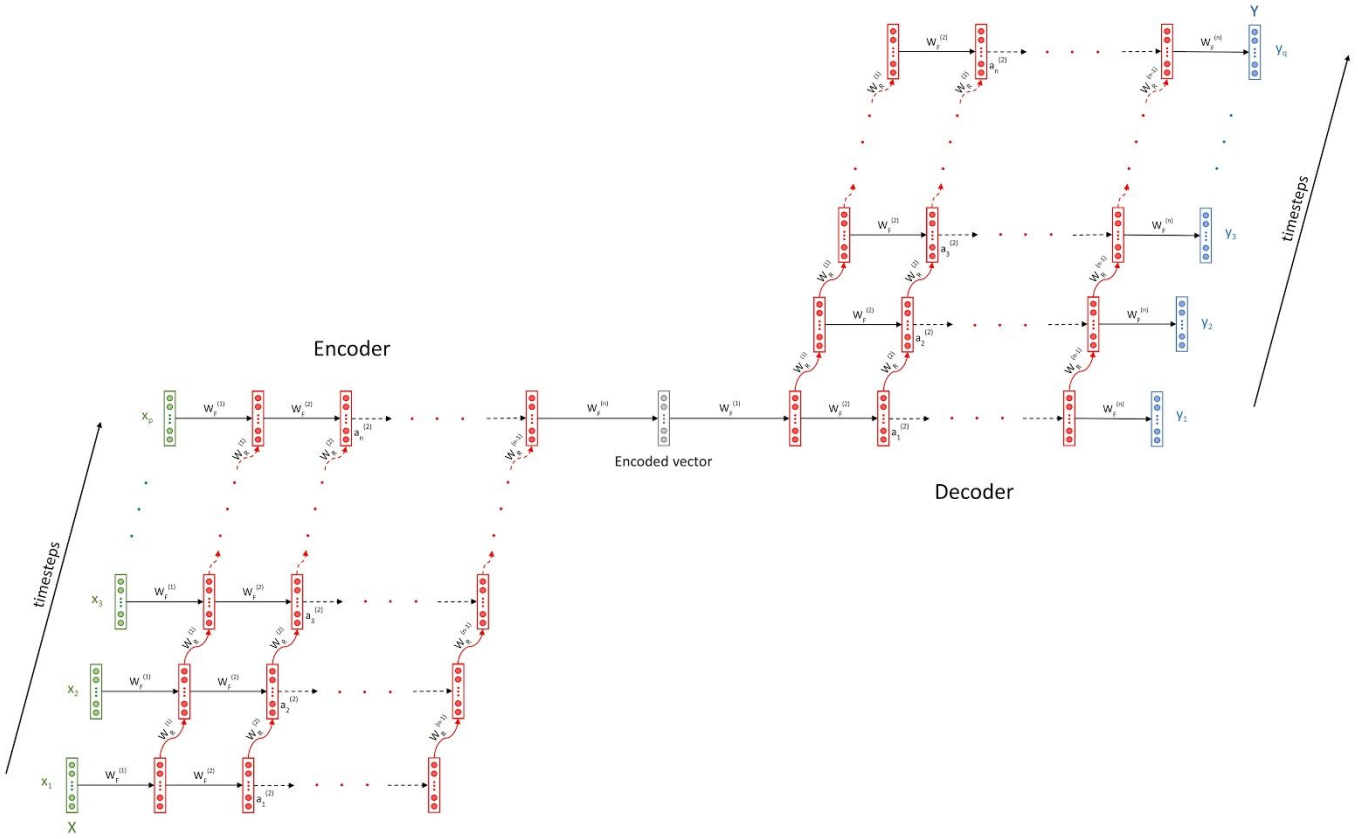


Figure 4: An encoder-decoder architecture of an RNN

You learnt that this is also a **many-to-many architecture** type. But the input and output sequences don't have a one-to-one correspondence. As a result, most often than not, the **length of the input and the output sequence is not equal**. You learnt that this architecture can be deployed in problems such as language translation and document summarization. You also learnt that the errors are backpropagated from the decoder to the encoder. The encoder and decoder have a different set of weights and they are different RNNs altogether. The loss is calculated at each timestep which can either be backpropagated at each timestep, or the cumulative loss (sum of all the losses from all the timesteps of a sequence) can be backpropagated after the entire sequence is ingested. Generally, the errors are backpropagated once an RNN ingests an entire batch.

- **One-to-many architecture**:

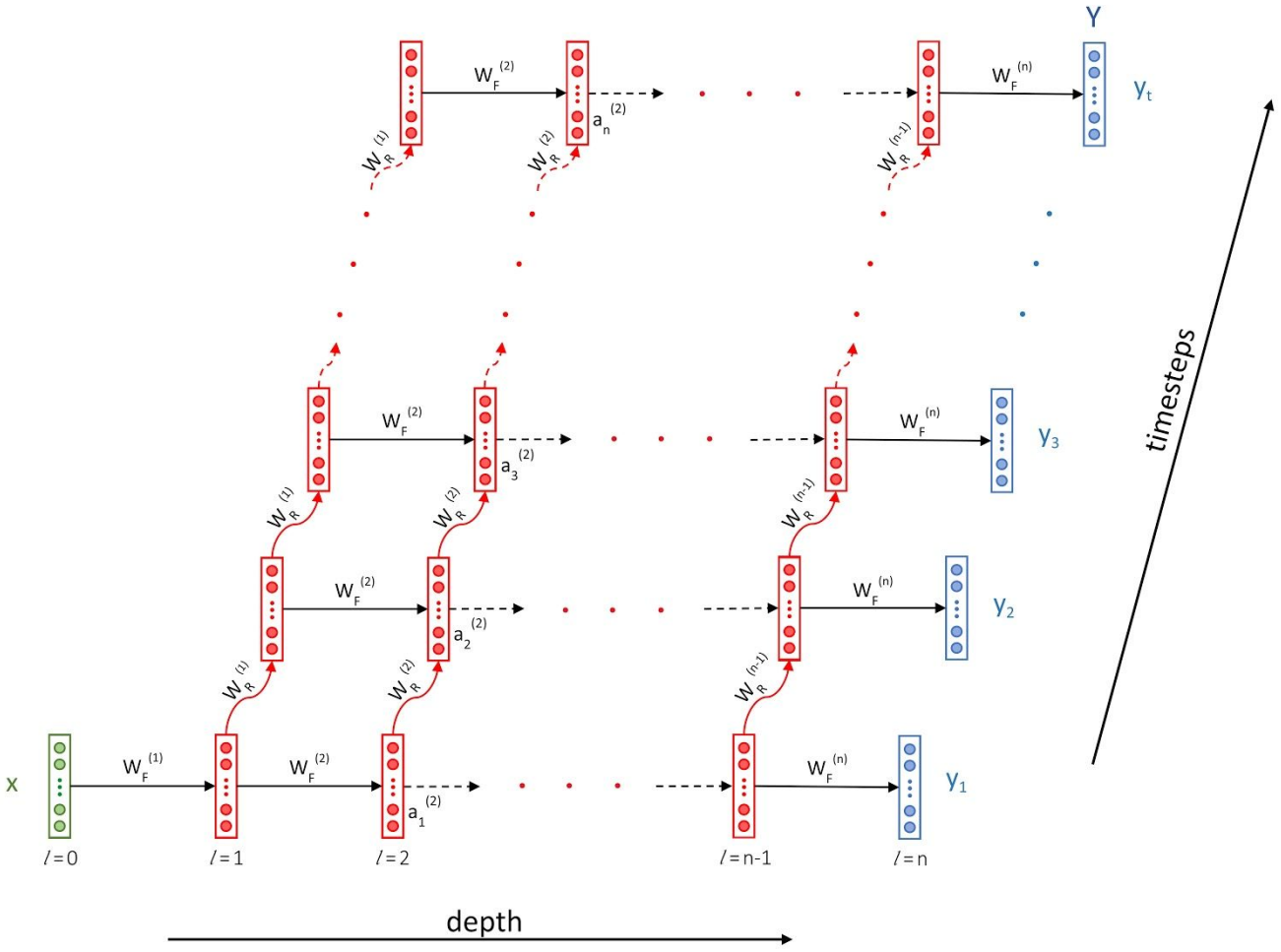


Figure 5: A one-to-many architecture of an RNN

You learnt that this type of architecture has a **single entity as an input and a sequence as the output**. You can use this architecture for generation such as music generation, creating drawings, generating text, etc.

Backpropagation Through Time (BPTT)

After going through the architectures, you learnt the mechanism in which gradients flow in an RNN. This mechanism is called **backpropagation through time** (BPTT). You learnt that in any given term in an RNN depends not only on the **current input** but also on the **input from previous timesteps**. The gradients not only flow back from the output layer to the input layer, but they also flow back in time from the last timestep to the first timestep. Hence the name backpropagation through time.

You also learnt that the backpropagation problem leads to the problem of **vanishing and exploding gradients**. The reason being the recurrent weight matrix W_R. W_R is raised to higher and higher power the more one goes back in time.

Bidirectional RNNs

You learnt that there are two kinds of problems in sequences:

- **Offline sequences**: These allow for a lookahead in time. The entire sequence is present for your perusal. For example, a document present in your local drive where you have access to the entire document.
- **Online sequences**: These don't allow for a lookahead. For example, a person is speaking and you need to transcribe the speech. In this case, you don't know what is going to come next.

You learnt that you can make use of offline sequences by looking ahead. You can feed the offline sequences to an RNN in **regular order as well as the reverse order** to get better results in whatever task you're doing. Such an RNN is called a **bidirectional RNN**. You also learnt that in a bidirectional RNN, the input at each timestep is a concatenation of the entity present in regular order and the entity present in reverse order. For example, for a sentence of length 100, the input at the first timestep will be a concatenation of the first word x_1 and the last word x_{100}. You learnt that a bidirectional RNN has **2x number of parameters** than a vanilla RNN.

Long Short-term Memory (LSTM) Networks

To get rid of the vanishing gradients problem, you learnt that researchers came up with another type of cell that can be used inside an RNN layer, called the LSTM cell.

You learnt that the LSTM cell has **three characteristics**:

- **Cell state**: You learnt that the LSTM cell has an explicit '**memory**' which is stored in the cell state. This memory helps an RNN to store important information about the sequences that it ingests.
- **Gating mechanisms**: The gating mechanisms allow an LSTM cell to manipulate the cell state in an efficient way. It also allows an LSTM to output efficiently.
- **Constant error carousel**: You learnt that the errors flow smoothly without vanishing during backpropagating through time. You learnt that the errors can flow from the current cell state to the previous cell state uninterrupted. This helps the LSTM to learn long-term dependencies unlike a vanilla RNN.

Next, you saw how an LSTM cell looks like and what are its feedforward equations. A diagram of an LSTM cell is shown below.

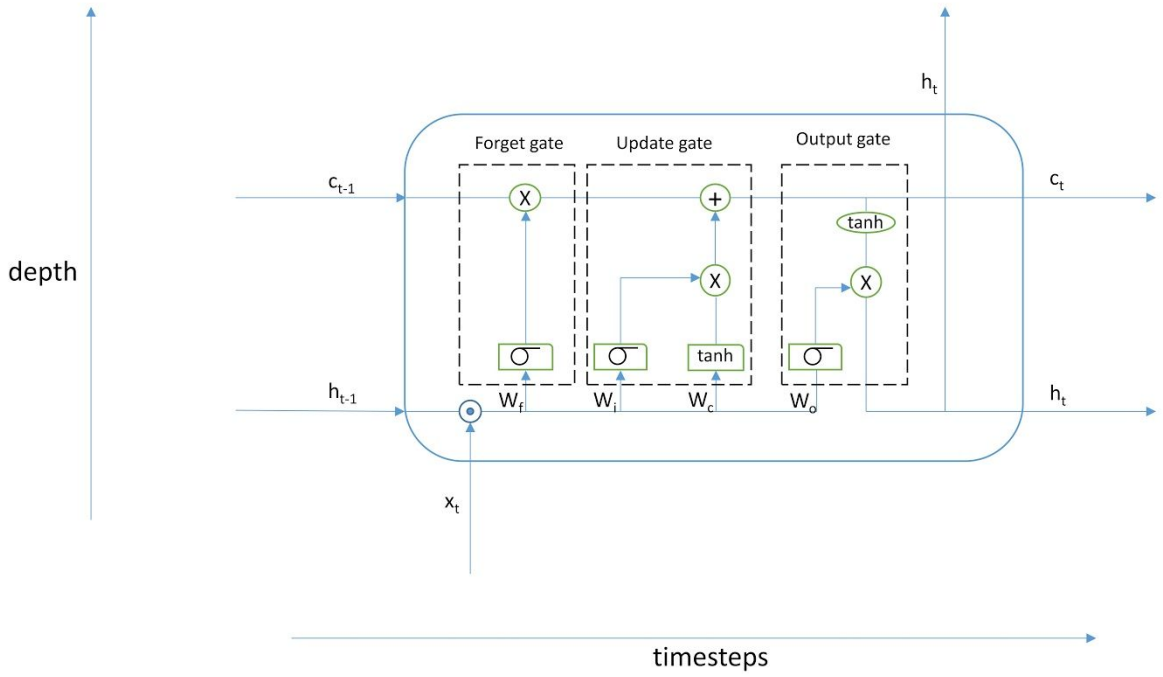


Figure 6: An LSTM cell

There are **three inputs** to an LSTM cell: the cell state from the previous timestep (c_{t-1}), the activation from the previous timestep (h_{t-1}) and the input from the current timestep (x_t) from the previous layer. There are **two outputs**: the current cell state (c_t) and the current activation (h_t) which goes in two directions: into the next timestep and into the next layer just like normal RNN activations are passed in two directions.

You also learnt that there are **three gates**: forget gate, update gate and the output gate. The forget gate is used to discard information from the previous cell state. The update gate writes new information to the previous cell state. After discarding and writing new information, you get the new cell state.

You learnt that an **LSTM layer** is made of multiple LSTM cells and an **LSTM network** can have multiple LSTM layers stacked on top of each other in the same way as an RNN with multiple RNN layers. The **feedforward equations** of an LSTM network are:

$$f_t = \sigma(W_f \left[h_{t-1}, x_t\right] + b_f)$$
$$i_t = \sigma(W_i \left[h_{t-1}, x_t\right] + b_i)$$
$$c'_t = tanh(W_c \left[h_{t-1}, x_t\right] + b_c)$$

$$c_t = f_t c_{t-1} + i_t c_t^{'}$$
$$o_t = \sigma(W_o \left[h_{t-1}, x_t\right] + b_o)$$
$$h_t = o_t \, tanh(c_t)$$

You also learnt that each of the fours weight matrices involved in the LSTM feedforward equations is a column-wise concatenation of the feedforward weight (W_F) and the recurrent weights (W_R) in the layer.

$$W_f = \left[W_{Ff} \mid W_{Rf}\right]$$
$$W_i = \left[W_{Fi} \mid W_{Ri}\right]$$
$$W_c = \left[W_{Fc} \mid W_{Rc}\right]$$
$$W_o = \left[W_{Fo} \mid W_{Ro}\right]$$

You learnt that as a result of 4 weight matrices and biases, an LSTM layer has **4x parameters** than an RNN layer.

Gated Recurrent Unit (GRU) Networks

Finally, you briefly saw an LSTM variant - the **gated recurrent unit** (GRU). A GRU network consists of GRU layers which consist of GRU cells which are similar to LSTM cells. However, the GRU has fewer parameters than an LSTM network. A GRU has three weight matrices as compared to the four in an LSTM layer. This means that a GRU has **3x parameters** than a vanilla RNN layer.

RNNs in Python

Finally, you learnt how to build different types of RNN in Python using the Keras library. You are recommended to go through the RNN code provided to you.

Disclaimer: All content and material on the UpGrad website is copyrighted material, either belonging to UpGrad or its bonafide contributors and is purely for the dissemination of education. You are permitted to access print and download extracts from this site purely for your own education only and on the following basis:

- You can download this document from the website for self-use only.
- Any copies of this document, in part or full, saved to disc or to any other storage medium may only be used for subsequent, self-viewing purposes or to print an individual extract or copy for non-commercial personal use only.
- Any further dissemination, distribution, reproduction, copying of the content of the document herein or the uploading thereof on other websites or use of content for any other commercial/unauthorized purposes in any way which could infringe the intellectual property rights of UpGrad or its contributors, is strictly prohibited.
- No graphics, images or photographs from any accompanying text in this document will be used separately for unauthorised purposes.
- No material in this document will be modified, adapted or altered in any way.
- No part of this document or UpGrad content may be reproduced or stored in any other web site or included in any public or private electronic retrieval system or service without UpGrad's prior written permission.
- Any rights not expressly granted in these terms are reserved.