Long Short-Term Memory Network 5

Sequential events have tremendous impacts in real life whereas conventional feedforward neural network (FNN) based approaches are not appropriate to handle such events. Recurrent Neural Network (RNN) is the pioneer model to handle sequential data which is an extension of FNN with special loop for information persistence. At present, Long Short-Term Memory (LSTM) Network with special gating units instead of neuron units is the most successful deep learning method for sequential data.

The rest of the chapter is organized as follows. Section 5.1 briefly explains the motivation of LSTM development. Section 5.2 presents different aspects of sequential data. Section 5.3 describes RNN and its limitations to handle sequential data. Section 5.4 explains LSTM architecture describing its features and individual components. Section 5.5 explains training of LSTM in detail. Section 5.6 elaborates LSTM for two sample problems: sequence prediction and pattern classification. Finally, MATLAB implementation of LSTM is discussed in Section 5.7.

5.1 Motivation

In decision making, human not only relies on present observations or elements but also considers previous experiences regarding the matter. Every time we get new information, the brain stores it for future context. This is a very powerful way to learn and this is what makes our brain special. Our brain is trained to learn from

a sequence of events and not necessarily just a single instance. In other words, brain does not start thinking from scratch for an event but performs in a persistence manner. As an example, one understands each word based on understanding of previous words while reading something. Finally, life is a sequence of steps; and therefore, for same events in different time steps the action may change based on previous encountered experiences.

Sequential events have tremendous impacts on computational intelligence paradigm. Classical feedforward neural network (FNN) based approaches, which are explained in the previous chapters, are not appropriate for time dependent sequential matter. A glaring limitation of FNNs (including convolutional neural networks) is that they are too constrained: they accept a fixed-sized vector as input (e.g., an image) and produce a fixed-sized vector as output (e.g., probabilities of different classes). Not only that, these models perform this mapping using a fixed number of computational steps (e.g., the number of layers in the model). The recurrent neural networks (RNNs), extending FNN with special loop for information persistence, developed to handle sequential data. But training of RNNs encounter several problems such as vanishing gradients. At present, Long Short-Term Memory Networks, with special gating units instead of neuron units, is the most successful deep learning method for sequential data.

5.2 Aspects of Sequential Data

Countless learning tasks require dealing with sequential data. Some of the commonly occurring sequences in real life include video, speech, stock market, sensor data, and so on. Sequences have complex

temporal dependencies and a lot of hidden information. Image captioning, speech synthesis, and music generation all require that a model produces outputs that are sequences. In other domains, such as time series prediction, video analysis, and musical information retrieval, a model must learn from inputs that are sequences. Interactive tasks, such as translating natural language, engaging in dialogue, and controlling a robot, often demand both capabilities. From computational point of view, the task with sequential data is termed as sequential learning which is broadly classified into two categories: sequence labeling and time-series prediction. In sequence labeling, a sequence of labels is applied to a sequence of data and its applications include speech recognition, gesture recognition, protein secondary structure prediction, handwriting recognition. On the other hand, the history of a time series is used to predict the next point in time-series prediction task. Applications of the task include stock market prediction, weather forecasting, object tracking, disaster prediction.

There are several aspects of computational models with sequential data. Fig. 5.1 demonstrates sequential events in different categories based on types of input and output in different sequence manners. In figure each rectangle is a vector and arrows represent functions (e.g., matrix multiply). Input vectors, output vectors and hidden state are marked with *I*, *O* and *H*, respectively. From left to right: (a) fixed-sized input to fixed-sized output. It is ordinary classification (e.g., image classification) (b) Sequence output for fixed size input. Sample task in this category is image captioning from

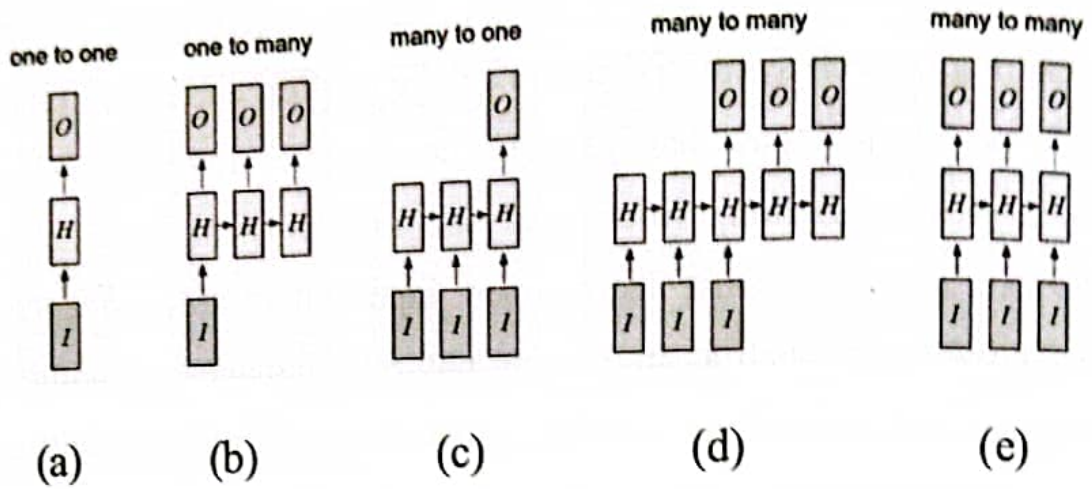


Figure 5.1: Aspects of computational models with sequential data.

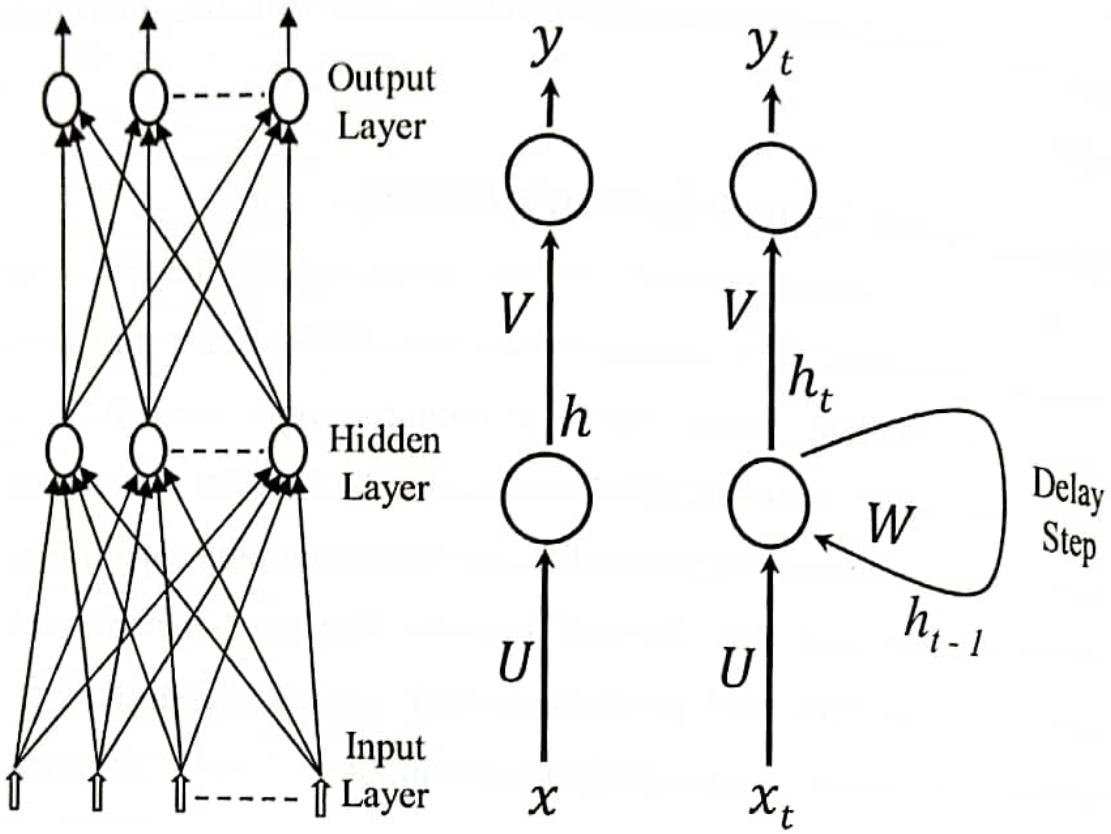
a fixed-size image where output is a sentence with a sequence of words. (c) Sequence input and single vector fixed output. The task in this category includes sentiment analysis where a given sentence is classified as expressing positive or negative sentiment. (d) Sequence input and sequence output. Machine translation is such kind of task where system reads a sentence in a language (e.g., English) and then outputs a sentence in another language (e.g., French). (e) Synced sequence input and output. An example of task in this category is labeling of each frame in a video sequence. Multilayer perceptron (MLP), Convolutional neural network (CNN) or any other model explained in the previous chapters are only capable of handling fixed-sized input to fixed-sized output problem (category a in Fig. 5.1) where output depends on input only through processing by the hidden state. Output in other categories depends on not only present input but also previous states. In such cases, no pre-specified constraints on the lengths of sequences because the recurrent transformation (in the middle) is fixed and can be applied as many times recurrently. A

different architecture is necessary to handle such real life important tasks.

5.3 Recurrent Neural Network (RNN)

RNN is a neural network model proposed in the 80's for modelling sequential data, such as time series. RNN architectures can have many different forms and one common type consists of a feedforward neural network (FNN) (i.e., standard MLP) plus added loops. These can exploit the powerful non-linear mapping capabilities of FNN and also have some form of memory. The loop(s) enables the networks to do temporal processing and learn sequences, e.g., perform sequence recognition/reproduction or temporal association/prediction.

Figure 5.2 shows the basic RNN formation with FNN adding recurrent loop with a delay unit in the hidden layer. Fig. 5.2(a) shows basic three layer FNN and its representation in matrix notation form for better explanation RNN. As it is already discussed, FNN generates hidden layer output vector h from input vector x through weight set U and then generates network output y from vector h through weight set V. FNN processes every input independently and generates output (i.e., y) regardless the status of previous step. Therefore, FNN's output is step/time independent, i.e., output is always same for a particular input pattern. On the other hand, output of RNN (Fig. 5.2(b)) is time dependent due to the delay step loop. The delay unit holds activations until they are processed at the next time step. RNN's hidden layer considers response of its previous step (i.e., h_{t-1}) as an input along with present input x_t. The delay step does not process anything but h_t



(a) Feedforward NN model (b) Recurrent NN model

Figure 5.2: Basic recurrent neural network model with recurrent loop in hidden nodes of feed forward NN.

becomes h_{t-1} for coming step feeding through a separate weight set W which maintain full connection in hidden layer units. Therefore, hidden layer output of RNN not only depends on the present input but also with its past hidden layer status i.e., h_{t-1}. Consequently, RNN output is time dependent, output y_t not only depends on the present input x_t but also with its past hidden layer activity h_{t-1}. Hidden layer output with activation function $\phi(.)$ is shown Eq. (5.1) where bias (b_h) is not presented for simplicity.

$$h_t = \phi(\hat{h}) = \phi(Ux_t + Wh_{t-1}). \tag{5.1}$$

Then RNN output with activation function $\sigma(.)$ without mentioning bias(b_o) is

$$y_t = \sigma(\hat{y}) = \sigma(Vh_t). \tag{5.2}$$

In general *tanh* and sigmoid functions are considered as the activation function in hidden and output layers, respectively. Since h_{t-1} influences output y_t through hidden layer output h_t without input (x_t = 0) output may alter due to only h_{t-1}.

The RNN can be converted into a FNN by unfolding over time. In RNN, activity of time t pass through time $t+1$. Thus, a RNN may be unrolled in time into a FNN as of Fig. 5.3. The activity of time $t+1$ not only depends on input x_{t+1} but also the unit activities of previous times step. Even though a RNN may unroll in time to make it mimic a FNN there is something wrinkle that make training slightly different. In the unrolled network, weights U, V and W are shared across the different layers through time. It may think the hidden state h_t as the

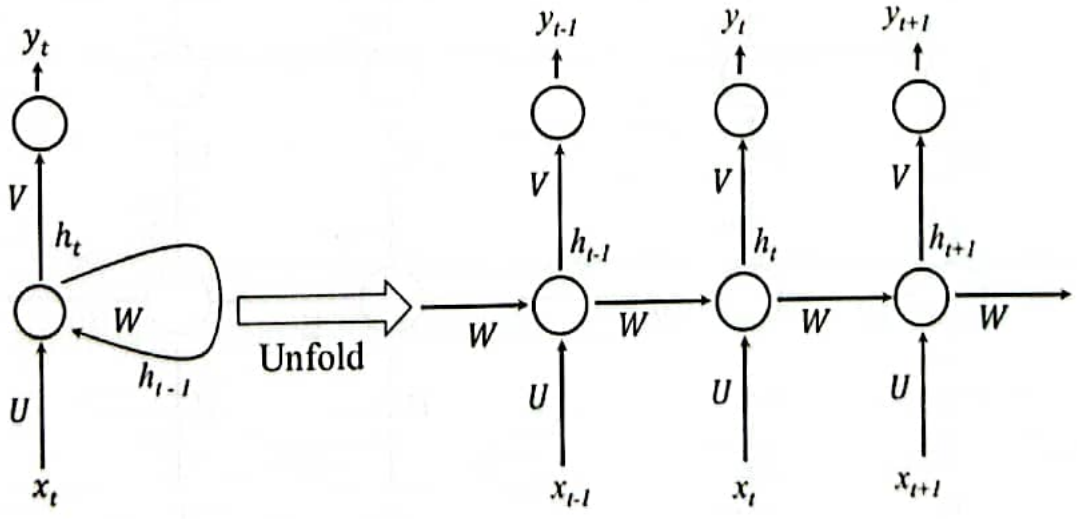


Figure 5.3: Unfolding RNN to FNN.

memory of the network and captures information about what happened in all the previous time steps. The output at step y_t is calculated solely based on the memory at time t. As briefly mentioned above, it's a bit more complicated in practice because h_t typically cannot capture information from too many time steps ago.

5.3.1 RNN Training

RNN training is a little complicated than FNN although it eventually unfolds into FNN. For better understanding of training operation, Fig. 5.4 is an unrolled RNN from beginning to time step t with error assignment comparing the generated output with corresponding desired outputs for each time step. Unrolled RNN may be considered as a deep FNN with t layers; difference is that FNN uses different parameters in each layer but RNN shares U, V and W.

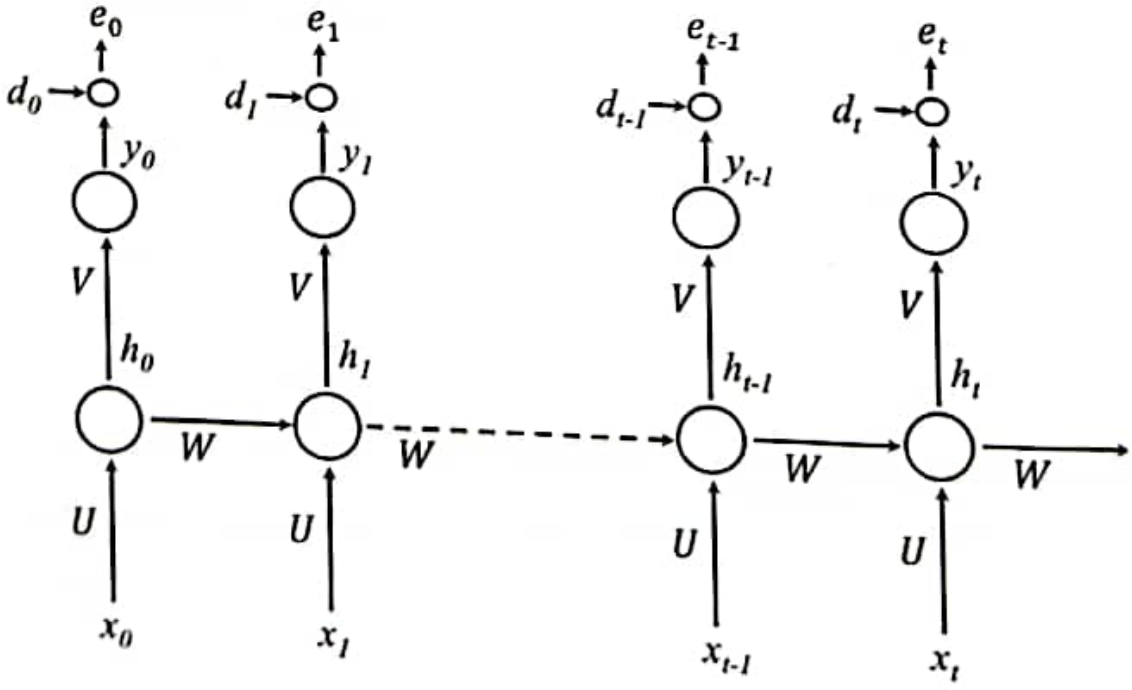


Figure 5.4: Sequence of error in unfolded RNN.

This reflects that RNN performs same task at each step, just with different inputs. This gradually reduces the total number of parameters needed to learn but made little complication at the same time due to weight sharing. In training with sequential data, desired output may alter for every input step (e.g., time series prediction) or retain same for a sequence of inputs (e.g., sequence labeling or sentiment analysis). In first case, network is updated for every input as desired output change with every input. In second case, full sequence (i.e., complete sentence or paragraph in sentiment analysis) is considered as one training example, so the total error is just the sum of the errors at each time step (i.e., word).

Backpropagation Through Time (BPTT) (Jaeger, 2002; Guo, 2013; Chen 2016) is considered for training RNN which is a natural extension of standard Backpropagation (BP) explained in Chapter 1. BPTT performs gradient descent on a complete unfolded network; i.e., unfolded for complete sequence. If a training sequence starts at time $t=0$ and ends at time $t=T$, the total cost function (E) is simply the sum over time of the standard error function e_t at each time step:

$$E(0,T) = \sum_{t=0}^{T} e_t \,, \tag{5.3}$$

and the gradient descent weight updates have contributions from each time step for weights U, V and W.

$$\Delta U = -\eta \frac{\partial E}{\partial U} = -\eta \sum_{t=0}^{T} \frac{\partial e_t}{\partial U} \tag{5.4}$$

$$\Delta V = -\eta \frac{\partial E}{\partial V} = -\eta \sum_{t=0}^{T} \frac{\partial e_t}{\partial V} \tag{5.5}$$

$$\Delta W = -\eta \frac{\partial E}{\partial W} = -\eta \sum_{t=0}^{T} \frac{\partial e_t}{\partial W} \tag{5.6}$$

Partial derivate for V, (i.e., $\partial e_t/\partial V$) is straight forward and depends on present input x_t and present hidden state h_t.

$$\frac{\partial e_t}{\partial V} = \frac{\partial e_t}{\partial y_t}\frac{\partial y_t}{\partial \hat{y}_t}\frac{\partial \hat{y}_t}{\partial V} = \frac{\partial e_t}{\partial y_t}\frac{\partial y_t}{\partial \hat{y}_t}\frac{\partial (Vh_t)}{\partial V} = \frac{\partial e_t}{\partial y_t}\frac{\partial y_t}{\partial \hat{y}_t} h_t \tag{5.7}$$

The constituent partial derivatives $\partial e_t/\partial U$ and $\partial e_t/\partial W$ now have contributions from the multiple instances of each weight U and W; and depend on the inputs and hidden unit activations at previous time steps. The errors now have to be backpropagated through time as well as through the network.

$$\frac{\partial e_t}{\partial U} = \frac{\partial e_t}{\partial y_t}\frac{\partial y_t}{\partial \hat{y}_t}\frac{\partial \hat{y}_t}{\partial h_t}\frac{\partial h_t}{\partial U} \tag{5.8}$$

$$\frac{\partial h_t}{\partial U} = \frac{\partial h_t}{\partial \hat{h}}\frac{\partial \hat{h}_t}{\partial U} = \phi'(t)\frac{\partial (Ux_t + Wh_{t-1})}{\partial U} = \phi'(t)\left(x_t + W\frac{\partial h_{t-1}}{\partial U}\right) = \phi'(t)\left(x_t + W\phi'(t-1)\left(x_{t-1} + W\frac{\partial h_{t-2}}{\partial U}\right)\right) \tag{5.9}$$

Again, for $\partial e_t/\partial W$

$$\frac{\partial e_t}{\partial W} = \frac{\partial e_t}{\partial y_t}\frac{\partial y_t}{\partial \hat{y}_t}\frac{\partial \hat{y}_t}{\partial h_t}\frac{\partial h_t}{\partial W} \tag{5.10}$$

$$\frac{\partial h_t}{\partial W} = \frac{\partial h_t}{\partial \hat{h}_t}\frac{\partial \hat{h}_t}{\partial W} = \phi'(t)\frac{\partial (Ux_t + Wh_{t-1})}{\partial W} = \phi'(t)W\frac{\partial h_{t-1}}{\partial W} = \phi'(t)W\phi'(t-1)W\frac{\partial h_{t-2}}{\partial W} \tag{5.11}$$

It is clear from the above weight update derivatives that the unfolded network is quite complex. Keeping track of all the

components at different points of time can become unwieldy. Typically, the updates are made in an online fashion with the weights being updated at each time step. This requires storage of the history of the inputs and past network states at earlier times. To be computationally feasible, truncation at a certain number of time steps, with the earlier information being ignored, is considered and called truncated BPTT.

Assuming the network is stable, the contributions to the weight updates should become smaller the further back in time they come from. This is because they depend on higher powers of small feedback strengths (corresponding to the sigmoid derivatives multiplied by the feedback weights). This means that truncation is not as problematic as it sounds, though many times step may be needed in practice (e.g., ~30). Despite its complexity, BPTT has been shown many times to be an effective learning algorithm.

Truncating the unfolded network to just one time step reduces it to a so-called Simple Recurrent Network which is also commonly known as an Elman network (Lipton et al., 2015). In this case, each set of weights now only appears only once, so it is possible to apply the gradient descent approach using the standard backpropagation algorithm rather than full BPTT. This means that the error signal will not get propagated back very far, and it will be difficult for the network to learn how to use information from far back in time. In practice, this approximation proves to be too great for many practical applications.

5.3.2 Difficulties in RNN Training

Vanishing and exploding gradients are common difficulties in training RNNs (Pascanu et al., 2013). It is notable that W is shared among all the different layers and thus gradients calculation involves previous steps as seen from Eq. (5.9) and Eq. (5.11). Therefore, for backpropagation the recursive operation will continue until $t=0$ across for every single time step. It is clearly observed from Eq. (5.11).

$$\frac{\partial h_t}{\partial W} = |W|^t \left|\frac{\partial h}{\partial \hat{h}}\right|^t = |W|^t |\phi'|^t \tag{5.12}$$

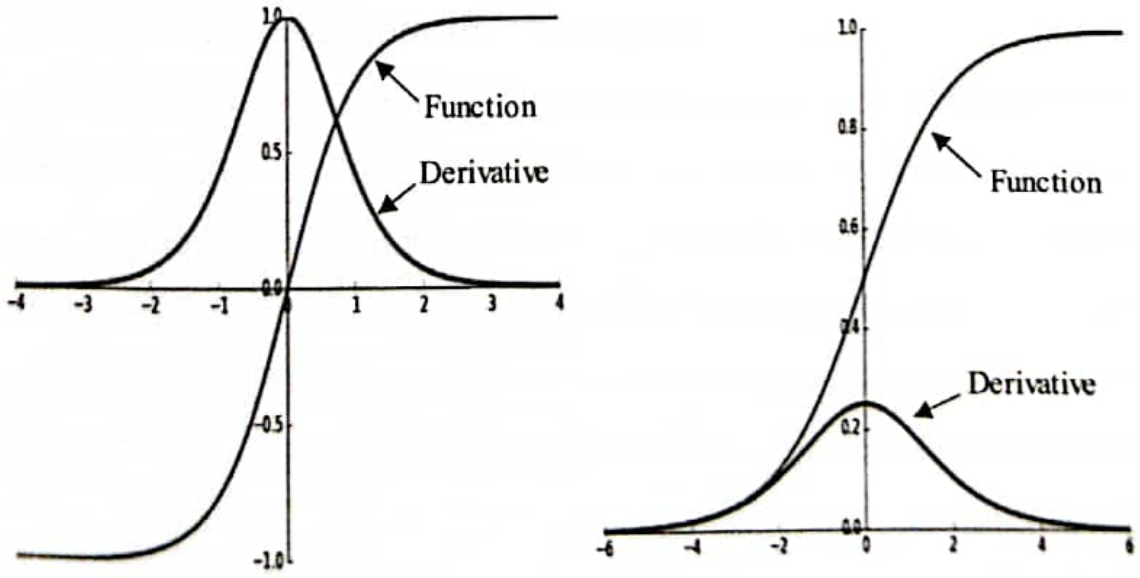
Considering the value of $\frac{\partial h_t}{\partial W}$ from Eq. (5.12) in Eq. (5.10)

$$\frac{\partial e_t}{\partial W} \propto |W|^t \left|\frac{\partial h}{\partial \hat{h}}\right|^t . \tag{5.13}$$

The Eq. (5.13) introduces the issue of vanishing and exploding gradients. Multiply a bunch of derivatives all the way back to $t=0$ in order to properly capture the influence of W. So then magnitude of update in scale with size of this weight matrix because of multiplying all the derivatives over and over again. If the recurrent weight W is less than 1 the gradient for long time (e.g., $t = 100$) going to near zero because all the weights with less than 1 are compounded together many times. This scenario is called vanishing gradients. Alternatively, if W is greater than 1 the issue is called exploding gradients where gradient step will be very large and may encounter incorrect step.

These are the common problems in training RNN especially for long sequence data. Similar effect also appears in calculating derivatives of input to hidden weight (i.e., U) as seen from Eq. (5.8) and Eq. (5.9).

The reasons of vanishing gradient matter may be explained with activation functions. Fig. 5.5 depicts functions and derivatives of *tanh* and *sigmoid* which are commonly used in hidden and output layers of RNN, respectively. *tanh* function maps all values into a range between -1 and 1, and the derivative is bounded by 1. On the other hand, *sigmoid* function maps all values into a range between 0 and 1, and the derivative is bounded by 1/4. Both *tanh* and *sigmoid* functions have derivatives of 0 at both ends. In both the ends, they approach a flat line and this happens when the corresponding neurons are saturated. At that point they have a zero gradient and drive other gradients in previous layers towards 0. Thus, with small values in the matrix and multiple matrix multiplications, the gradient values are



(a) tanh (b) sigmoid

Figure 5.5: tanh and sigmoid functions and their derivatives.

shrinking exponentially fast, eventually vanishing completely after a few time steps. Gradient contributions from "far away" steps become zero, and the state at those steps does not contribute to what are using to learn. Vanishing gradients are not exclusive to RNNs. They also happen in deep FNNs. It is just that RNNs tend to be very deep FNN which makes the problem a lot more common.

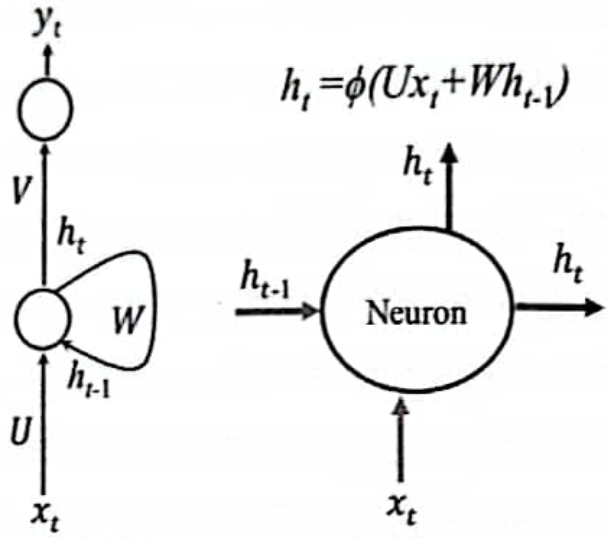
Vanishing gradients have received more attention than exploding gradients in RNN. Exploding gradients are obvious and clipping the gradients at a pre-defined threshold is a very simple and effective solution. On the other hand, vanishing gradients are more problematic because it is not obvious when they occur or how to deal with them. But through vanishing gradients the value will become NaN (not a number) and program might be crashed.

There are a few ways to combat the vanishing gradient problem. Proper initialization of the W matrix can reduce the effect of vanishing gradients. So can regularization. A more preferred solution is to use *ReLU* instead of *tanh* or *sigmoid* activation functions. The *ReLU* derivative is a constant of either 0 or 1, so it is not as likely to suffer from vanishing gradients. An even more popular solution is to use Long Short-Term Memory (LSTM) which architecture were explicitly designed to deal with vanishing gradients and efficiently learn long-range dependencies. Nowadays LSTM is the most successful model for sequential data handling in deep learning paradigm.

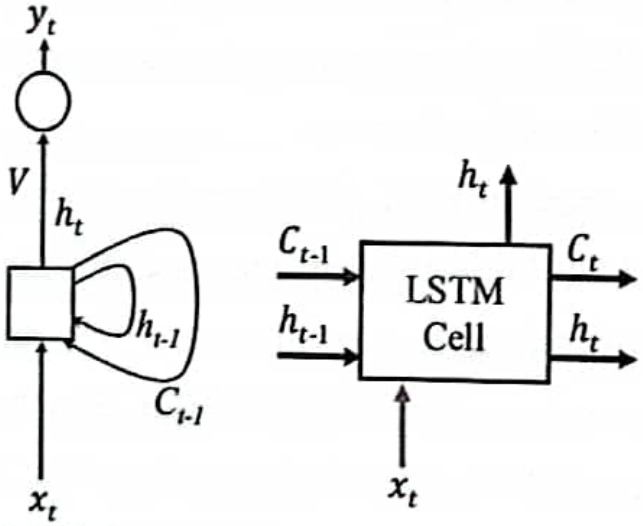
5.4 Long Short-Term Memory (LSTM) Network

Activities of individual steps in sequential data are inter related and output or response of the system depends on both present input and previous states. Therefore, the required main feature to handle sequential data is to remember previous states activities. RNN has ability to consider previous state activity in current decision through feedback loop in hidden layer, but its operation is uncontrolled. There is so far no constraints on the update states of RNN; its state knowledge can change pretty chaotically. This chaos means information quickly transforms and vanishes, and it is difficult for the model to keep a long-term memory.

LSTM is a variant of RNN that is capable of learning long term dependencies. LSTMs were first proposed by Hochreiter and Schmidhuber (Hochreiter & Schmidhuber, 1997); and refined by many other researchers (Greff et al., 2017). Fig. 5.6 presents a comparative view of RNN and LSTM network exploring hidden units of individual networks. The hidden unit in RNN is simple FNN neuron; present state (h_t) depends on previous state (i.e., h_{t-1}) and current input (x_t) as already explained. On the other hand, LSTM introduces long-term memory cell state (C_t) along with working memory (h_t). Memory cell state (C_t) is the core of an LSTM cell which is an additional recurrence element with respect to RNN. For an input x_t at a particular time step t, the cell updates C_{t-1} and h_{t-1} considering them as long-term memory and working memory, respectively. Finally, the network output (y_t) is taken from h_t as like RNN.



(a) RNN and neuron in hidden layer.



(b) LSTM network and cell in hidden layer.

Figure 5.6: Comparative demonstration of a RNN hidden neuron and a LSTM cell unit.

5.4.1 Details of LSTM Cell Unit

Few real memory features are conceived in LSTM development. Human brain resembles different types of memory and operations on the memory; long-term memory and short-term memory are two main operational memory. There are significant differences between short-term memory and long-term memory. Short-term memory gives the

ability for the human brain to keep information in the mind for a very short period of time and it also suffers from capacity. Long-term memory, however, can store vast amounts of information and is permanent. Information of short-term memory lasts within seconds to minutes then dissipates if effort is not made to retain long-term use consolidating in long-term memory bank. Things learned on a daily basis dissipate unless consolidation takes place. If short-term memory is consolidated and stored in the long-term memory, it becomes permanently ingrained and accessible for later use. It is believed that short-term memory is a primary function of the prefrontal cortex and long-term memories are formed and stored permanently in hippocampus.

The conversion from short-term to long-term memory requires concerted effort, the passage of time, and the absence of interference in memory consolidation. In addition, sleeping has been said to be important in the consolidation of memory and learning. Ultimately, the role of short-term memory is to file information for temporary usage. If it is not consolidated, it is discarded. This process of discarding is important to make room for learning and new memories. But once a memory is stored in the long-term memory bank, it is stored there forever. It may not feel like it is stored there forever. Sometimes, you may not be able to recall something that is stored in the long-term memory bank-nevertheless, it is there. Because once memory is stored, it is permanent. Finally, brain performs operations memorize, forget, consolidation into long-term memory for long term use and recall from the long-term memory.

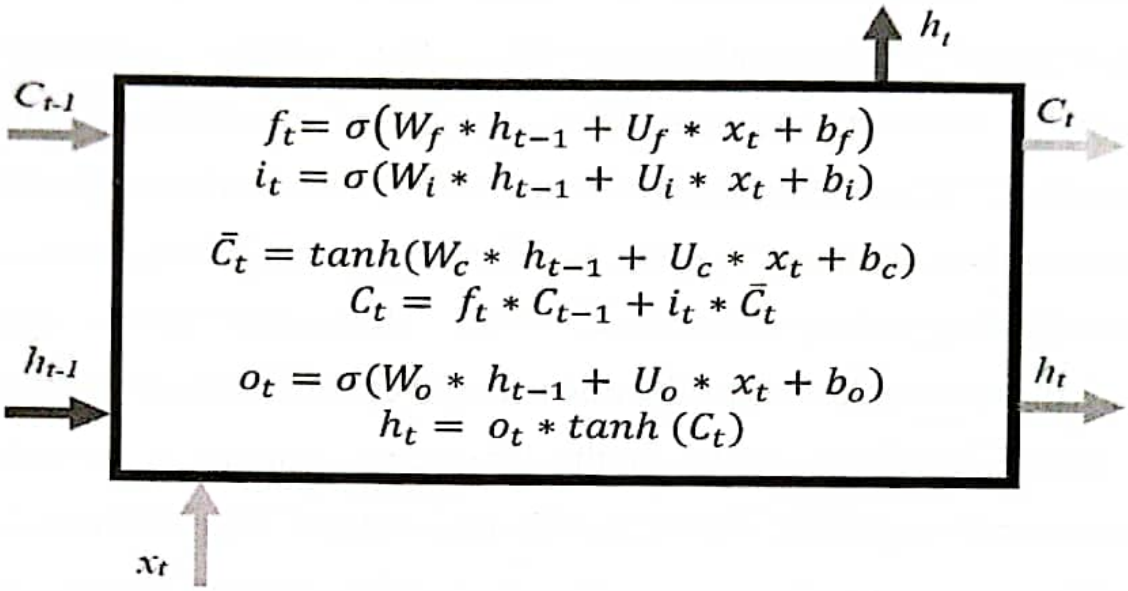
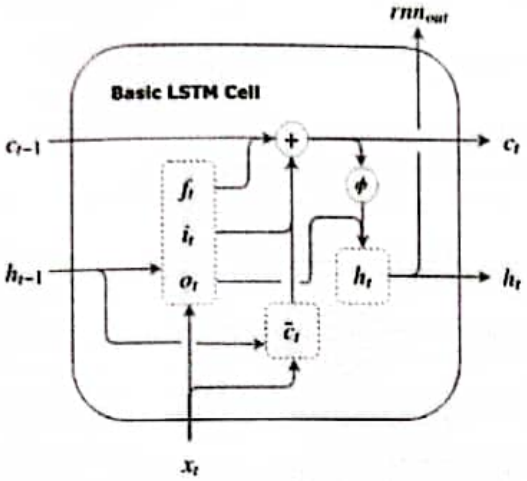


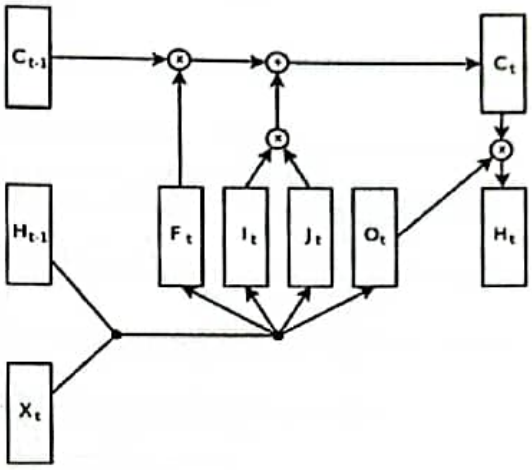
Figure 5.7: Internal cell operation equations of LSTM cell.

Figure 5.7 shows LSTM cell with the equations at a glance which are behind the LSTM cell operations. The memory cell unit allows an LSTM network to adaptively forget, memorize and expose the memory content with special operations. It is clear for the figure that LSTM operation is much complex having several *tanh* (symbol φ) and sigmoid (symbol σ) functions with respect to RNN which uses single *tanh* function to update its hidden state. Among the equations, sigmoid (i.e., σ) functions are used as gates. Each individual equation has significant purpose for memory operation.

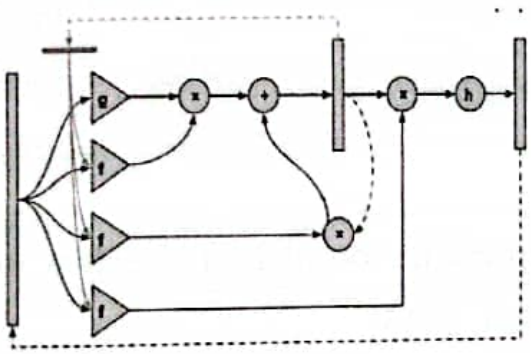
A number of different symbolic representations of the LSTM cell operation with the equations are demonstrated by different research groups; among those few prominent ones are shown in Fig. 5.8. Although the figures look significantly different but all represent the LSTM operational equations of Fig. 5.7.



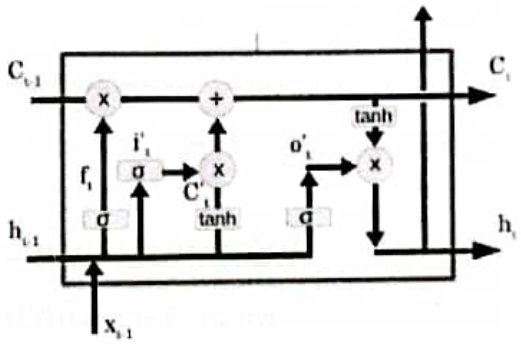
[WM, 2016]



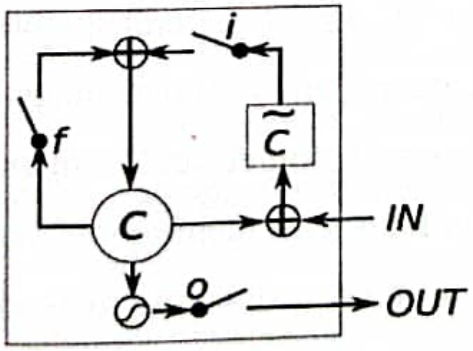
[Jozefowicz et al., 2015]



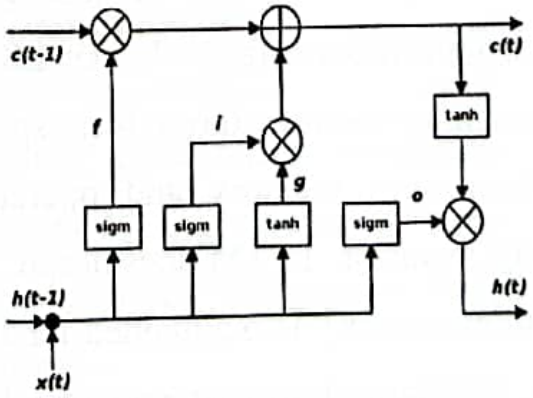
[Breuel, 2015]



[Bhat et al., 2016]



[Gakhov, 2015]



[Gulli and Pal, 2017]

Figure 5.8: Different diagrams of internal LSTM cell.

A typical symbolic representation of LSTM cell with marking individual state is shown in Fig. 5.9. LSTM network is the precise

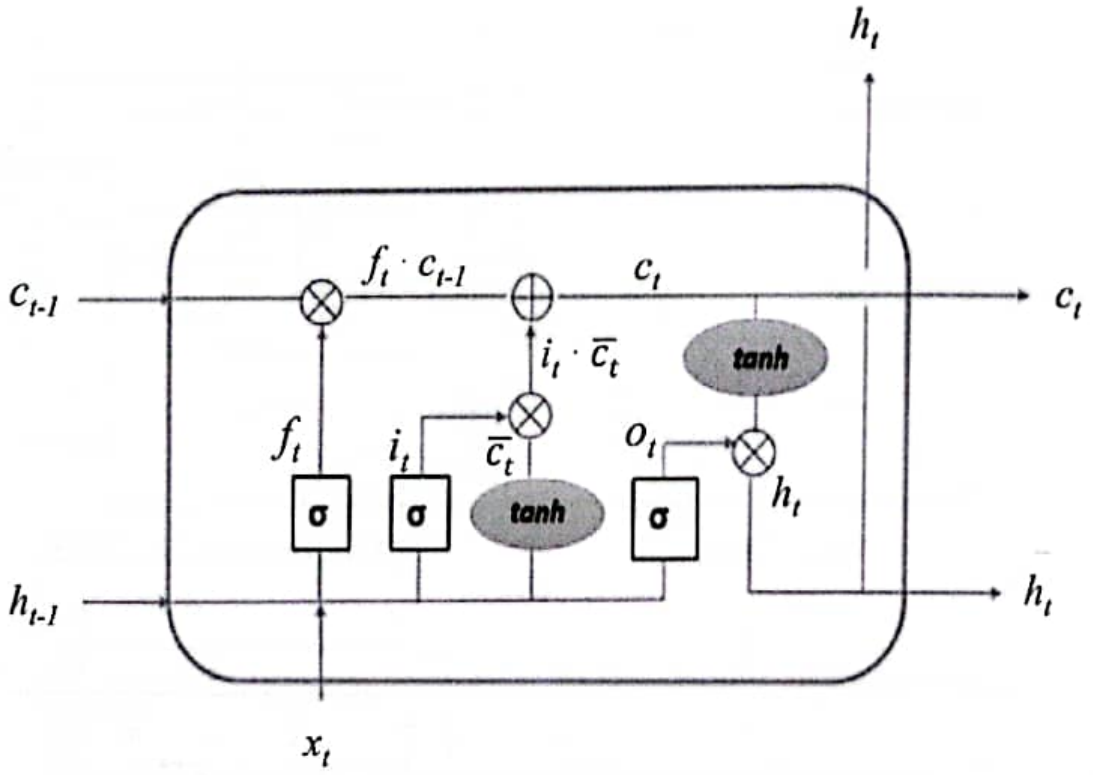


Figure 5.9: A typical symbolic representation of LSTM cell with marking individual state with [Colah, 2015].

memory like system to handle sequential events with long-term dependencies effectively. An LSTM unit mimics several important memory features through its special gating units. The features are short-term memory and operations on it forget and remember mechanisms. LSTM does not maintain long-term memory but short-term memory is maintained for long term uses; thus, the method so called long short-term memory. An LSTM cell take input sequence x_t and updates hidden state (i.e., h_t) and memory cell state (C_t) through its various operations. However, in several studies, C_t is defined as long-term memory and h_t is defined as working memory which is a kind of short-term memory. Based on Fig. 5.9, the individual operations with their underlying equations are explained below.

1. Forget/Remember Previous Memory Cell State:

An LSTM remembers required previous memory cell state information forgetting a few selected portions. The forget gate (f_t) with a sigmoid layer makes decision about which portion to remember and forgetting some selected potions. It takes h_{t-1} and x_t as input and provides an output between 0 and 1 for each number in the memory cell state C_{t-1}. If its' output is 1, it will keep the data completely, while for 0 it will forget the data.

$$\text{Forget Gate: } f_t = \sigma(W_f * h_{t-1} + U_f * x_t + b_f) \quad (5.14)$$

Point wise multiplication of C_{t-1} vector with f_t vector performs forget operation despite the fact that a 1 in f_t means to keep the memory point and a 0 still means to forget it. Finally, $f_t * C_{t-1}$ is the remembered memory from previous memory cell state and pass forward for further operations.

2. Memory Cell State Update with New Memory

An LSTM cell generates new memory manipulating h_{t-1} and x_t, and selective portions of the new memory are stored in the memory cell state. New candidate memory values $\bar{C}_t$ is generated with *tanh* layer. The input gate (i_t) with a sigmoid layer makes decision which potion of $\bar{C}_t$ will be stored in the memory.

Candidate New Memory:

$$\bar{C}_t = tanh(W_c * h_{t-1} + U_c * x_t + b_c) \quad (5.15)$$

$$\text{Input Gate: } i_t = \sigma(W_i * h_{t-1} + U_i * x_t + b_i) \quad (5.16)$$

Outcome of point wise multiplication of vector $\bar{C}_t$ with i_t vector

(i.e.,$i_t * \bar{C}_t$) is value to transfer memory cell. Finally, memory cell state is updated adding $i_t * \bar{C}_t$ with remembered memory (i.e., $f_t * C_{t-1}$).

Updated Memory Cell State: $C_t = f_t * C_{t-1} + i_t * \bar{C}_t$ (5.17)

3. Hidden State Update

In LSTM, hidden state (h_t) is exposed from memory cell state (C_t), and h_t is then used to produce cell output like RNN. h_t is the filtered version of C_t which is generated in two steps. In the first step, C_t is passed through a *tanh* to push the values to be between -1 to 1; and in the second step, output gate (o_t) is used to filter the values and hence produce h_t. Output gate (o_t) makes the assessment regarding what parts of the memory cell state C_t needs to be exposed/present in the hidden state h_t. Finally, h_t is the point wise multiplication of $tanh(C_t)$ and o_t.

Output Gate $o_t = \sigma(W_o * h_{t-1} + U_o * x_t + b_o)$ (5.18)

Updated Hidden State $h_t = o_t * tanh\,(C_t)$ (5.19)

4. Initialization and Cell Update without Input

At a glance, the cell first takes the advice of the forget gate f_t and accordingly forgets the past memory C_{t-1}. Similarly, it takes the advice of the input gate i_t and accordingly gates the new memory $\bar{C}_t$. It then sums these two results to produce the final memory C_t. Finally, produce hidden state h_t with the help of output gate o_t. All the tree gates manipulate h_{t-1} and x_t individually through different weights for decision outcomes. In initial condition, both h_{t-1} and C_{t-1} have no value,

then C_t and h_t depend on x_t simplifying Eqs. (14) – (18) as follows.

$$f_t = \sigma(U_f * x_t + b_f) \tag{5.20}$$

$$\bar{C}_t = tanh(U_c * x_t + b_c) \tag{5.21}$$

$$i_t = \sigma(U_i * x_t + b_i) \tag{5.22}$$

$$C_t = i_t * \bar{C}_t \tag{5.23}$$

$$o_t = \sigma(U_o * x_t + b_o) \tag{5.24}$$

On the other hand, if there is no input ($x_t = 0$) then LSTM cell will go to action and update different states due to previous step states C_{t-1} and h_{t-1} transferring the Eqs. (14)-(18) in the equations bellow.

$$f_t = \sigma(W_f * h_{t-1} + b_f) \tag{5.25}$$

$$\bar{C}_t = tanh(W_c * h_{t-1} + b_c) \tag{5.26}$$

$$i_t = \sigma(W_i * h_{t-1} + b_i) \tag{5.27}$$

$$C_t = f_t * C_{t-1} + i_t * \bar{C}_t \tag{5.28}$$

$$o_t = \sigma(W_o * h_{t-1} + b_o) \tag{5.29}$$

For both scenarios h_t comes from o_t and C_t same as Eq. (19) and

$$h_t = o_t * tanh(C_t) \ . \tag{5.30}$$

It is observed from the equations that output of the LSTM may alter in the proceeding time steps due to recurrent connection of C_t and h_t.

5.5 LSTM Network Training

The aim of training of any network is to get appropriate weight sets to perform specific tasks. This section explains training of a

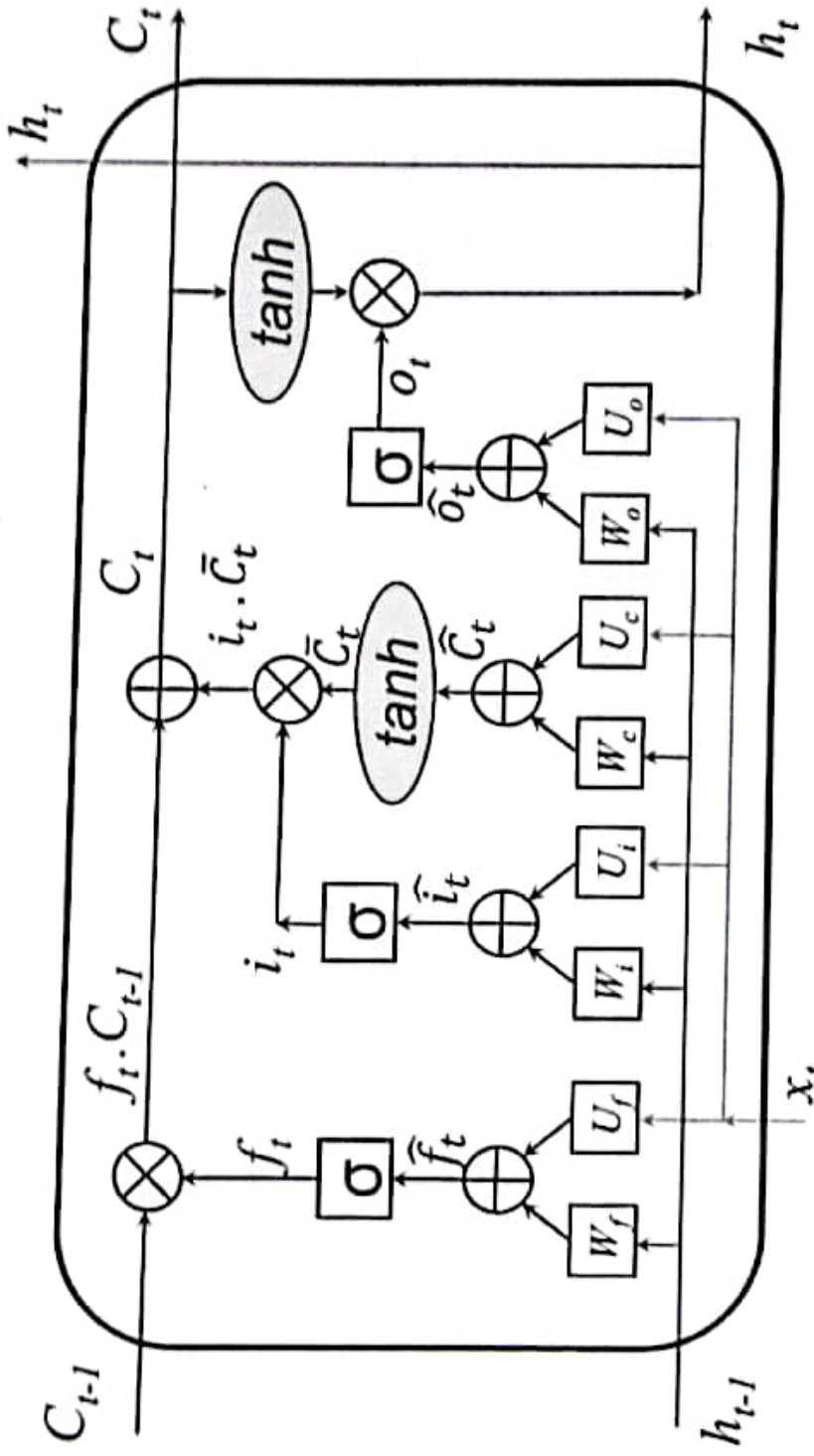


Figure 5.10: LSTM cell with weights in positions.

simple LSTM network. For better understanding, a detailed LSTM cell placing weight sets is shown in Fig. 5.10. On the other hand, direction of weight sets update is shown as dotted line in Fig. 5.11.

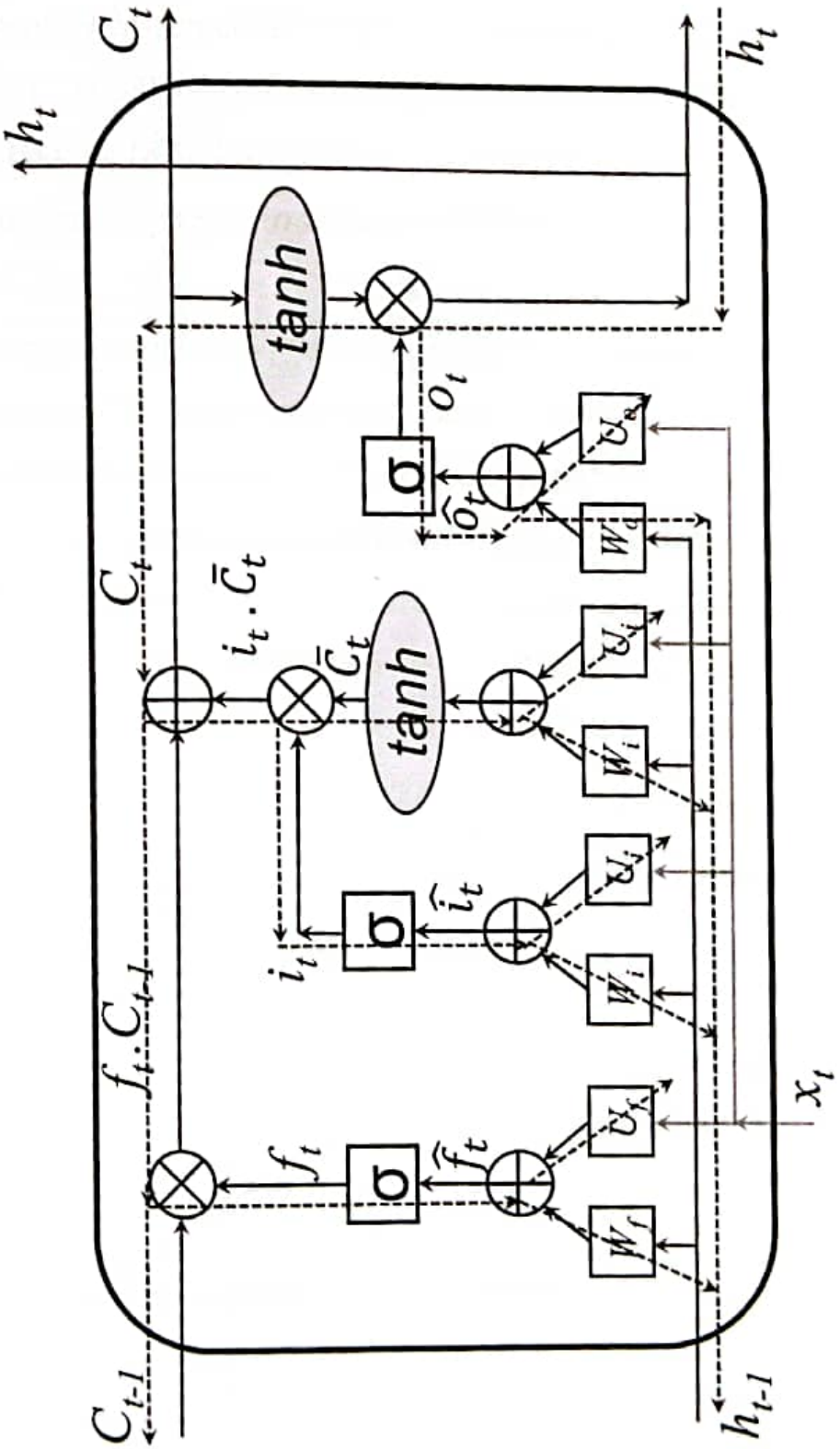


Figure 5.11: Weight update direction marked in dotted line on LSTM cell.

Figure 5.12 shows the network with a LSTM unit and unfolded for a sequence from initial point to time step t like RNN. In the network, output (y_t) is generated from LSTM output (h_t) through weight V like RNN. Other shared weights (i.e., W_f, U_f, W_c, U_c, W_i, U_i, W_o and U_o) are inside the LSTM block. Training of LSTM network to update the weights is similar to RNN but steps of operations are quite different.

In supervised manner training, weights are updated for a complete sequence after accumulation of individual time steps. If a training sequence starts at time t=0 and ends at time t=T, the total cost function (E) is simply the sum over time of the standard error function e_t at each time step and the gradient descent weight updates have contributions from each time step for weights.

$$E(0,T) = \sum_{t=0}^{T} e_t \tag{5.31}$$

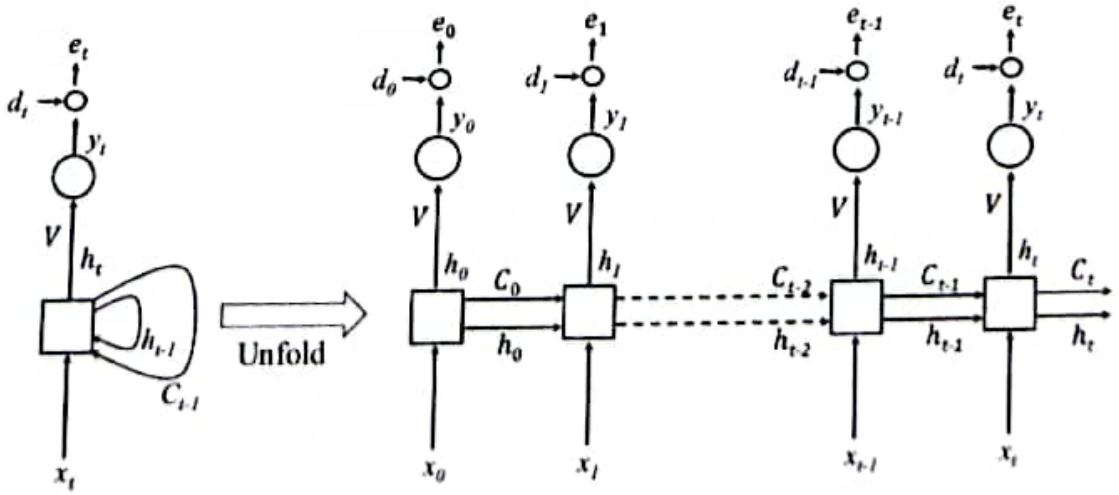


Figure 5.12: Simple LSTM network and unfolding from initial.

The update of weight V is same as RNN; partial derivate for V, (i.e., $\partial e_t/\partial V$) is straight forward and depends on present input x_t and present hidden state h_t.

$$\Delta V = -\eta \frac{\partial E}{\partial V} = -\eta \sum_{t=0}^{T} \frac{\partial e_t}{\partial V} \tag{5.32}$$

$$\frac{\partial e_t}{\partial V} = \frac{\partial e_t}{\partial y_t}\frac{\partial y_t}{\partial \hat{y}_t}\frac{\partial \hat{y}_t}{\partial V} = \frac{\partial e_t}{\partial y_t}\frac{\partial y_t}{\partial \hat{y}_t}\frac{\partial (Vh_t)}{\partial V} = \frac{\partial e_t}{\partial y_t}\frac{\partial y_t}{\partial \hat{y}_t}h_t \tag{5.33}$$

The derivatives of LSTM weights may be computed in two different steps. At first, derivatives for gates and states are calculated. Then derivatives of each individual weights are calculated considering the derivative of the corresponding gate or state. The sequence of derivatives calculation will be reverse of forward pass and hence the equation orders to get derivatives are as follows:

$$h_t = o_t * tanh\,(C_t) = o_t * \varphi(C_t) \tag{5.34}$$

$$o_t = \sigma(W_o * h_{t-1} + U_o * x_t + b_o) = \sigma(\hat{o}_t) \tag{5.35}$$

$$C_t = f_t * C_{t-1} + i_t * \bar{C}_t \tag{5.36}$$

$$\bar{C}_t = tanh(W_c * h_{t-1} + U_c * x_t + b_c) = \varphi(\hat{C}_t) \tag{5.37}$$

$$i_t = \sigma(W_i * h_{t-1} + U_i * x_t + b_i) = \sigma(\hat{i}_t) \tag{5.38}$$

$$f_t = \sigma(W_f * h_{t-1} + U_f * x_t + b_f) = \sigma(\hat{f}_t) \tag{5.39}$$

1. Derivatives for Gates and States

Derivative of Hidden State:

$$\frac{\partial e_t}{\partial h_t} = \frac{\partial e_t}{\partial y_t}\frac{\partial y_t}{\partial \hat{y}_t}\frac{\partial \hat{y}_t}{\partial h_t} = \frac{\partial e_t}{\partial y_t}\frac{\partial y_t}{\partial \hat{y}_t}.V \tag{5.40}$$

Derivative of Output Gate:

$$\frac{\partial e_t}{\partial o_t} = \frac{\partial e_t}{\partial h_t}\frac{\partial h_t}{\partial o_t} = \frac{\partial e_t}{\partial h_t}.\varphi(C_t) \tag{5.41}$$

Derivative of Memory Cell State:

$$\frac{\partial e_t}{\partial C_t} = \frac{\partial e_t}{\partial h_t}\frac{\partial h_t}{\partial C_t} = \frac{\partial e_t}{\partial h_t}.o_t.\varphi'(C_t) \tag{5.42}$$

Derivative of Input Gate and Forget Gate:

$$\frac{\partial e_t}{\partial i_t} = \frac{\partial e_t}{\partial C_t}\frac{\partial C_t}{\partial i_t} = \frac{\partial e_t}{\partial C_t}.\overline{C}_t \tag{5.43}$$

$$\frac{\partial e_t}{\partial f_t} = \frac{\partial e_t}{\partial C_t}\frac{\partial C_t}{\partial f_t} = \frac{\partial e_t}{\partial C_t}.C_{t-1} \tag{5.44}$$

Derivatives of New Memory and Previous Memory Cell State:

$$\frac{\partial e_t}{\partial \overline{C}_t} = \frac{\partial e_t}{\partial C_t}\frac{\partial C_t}{\partial \overline{C}_t} = \frac{\partial e_t}{\partial C_t}.i_t \tag{5.45}$$

$$\frac{\partial e_t}{\partial C_{t-1}} = \frac{\partial e_t}{\partial C_t}\frac{\partial C_t}{\partial C_{t-1}} = \frac{\partial e_t}{\partial C_t}.f_t \tag{5.46}$$

In LSTM $\dfrac{\partial C_t}{\partial C_{t-1}} = f_t$

Derivative of Previous Hidden State:

$$\frac{\partial e_t}{\partial h_{t-1}} = \frac{\partial e_t}{\partial o_t}\frac{\partial o_t}{\partial h_{t-1}} + \frac{\partial e_t}{\partial \bar{C}_t}\frac{\partial \bar{C}_t}{\partial h_{t-1}} + \frac{\partial e_t}{\partial i_t}\frac{\partial i_t}{\partial h_{t-1}} + \frac{\partial e_t}{\partial f_t}\frac{\partial f_t}{\partial h_{t-1}} \tag{5.47}$$

$$\frac{\partial e_t}{\partial h_{t-1}} = \frac{\partial e_t}{\partial o_t}.\sigma'(\hat{o}_t).W_o + \frac{\partial e_t}{\partial \bar{C}_t}.\varphi'(\hat{C}_t).W_c + \frac{\partial e_t}{\partial i_t}.\sigma'(\hat{i}_t).W_i + \frac{\partial e_t}{\partial f_t}.\sigma'(\hat{f}_t).W_f \tag{5.48}$$

2. Derivatives for Weights

Derivatives of Output Weights:

$$\frac{\partial e_t}{\partial W_o} = \frac{\partial e_t}{\partial o_t}\frac{\partial o_t}{\partial W_o} = \frac{\partial e_t}{\partial o_t}.\sigma'(\hat{o}_t).h_{t-1} \tag{5.49}$$

$$\frac{\partial e_t}{\partial U_o} = \frac{\partial e_t}{\partial o_t}\frac{\partial o_t}{\partial U_o} = \frac{\partial e_t}{\partial o_t}.\sigma'(\hat{o}_t).x_t \tag{5.50}$$

Derivatives of Memory Cell State Weights:

$$\frac{\partial e_t}{\partial W_c} = \frac{\partial e_t}{\partial \overline{C}_t}\frac{\partial \overline{C}_t}{\partial W_c} = \frac{\partial e_t}{\partial o_t}.\varphi'(\hat{C}_t).h_{t-1} \tag{5.51}$$

$$\frac{\partial e_t}{\partial U_c} = \frac{\partial e_t}{\partial \overline{C}_t}\frac{\partial \overline{C}_t}{\partial U_c} = \frac{\partial e_t}{\partial \overline{C}_t}.\varphi'(\hat{C}_t).x_t \tag{5.52}$$

Derivatives of Input Gate Weights:

$$\frac{\partial e_t}{\partial W_i} = \frac{\partial e_t}{\partial i_t}\frac{\partial i_t}{\partial W_i} = \frac{\partial e_t}{\partial i_t}.\sigma'(\hat{i}_t).h_{t-1} \tag{5.53}$$

$$\frac{\partial e_t}{\partial U_i} = \frac{\partial e_t}{\partial i_t}\frac{\partial o_t}{\partial U_i} = \frac{\partial e_t}{\partial i_t}.\sigma'(\hat{i}_t).x_t \tag{5.54}$$

Derivatives of Forget Gate Weights:

$$\frac{\partial e_t}{\partial W_f} = \frac{\partial e_t}{\partial f_t}\frac{\partial f_t}{\partial W_f} = \frac{\partial e_t}{\partial f_t}.\sigma'(\hat{f}_t).h_{t-1} \tag{5.55}$$

$$\frac{\partial e_t}{\partial U_f} = \frac{\partial e_t}{\partial f_t}\frac{\partial f_t}{\partial U_f} = \frac{\partial e_t}{\partial f_t}.\sigma'(\hat{f}_t).x_t \tag{5.56}$$

Outcomes of weight derivatives from Eqs. (5.49) to (5.56) are the crucial parts for weight updates to learn LSTM. The derivatives for gates and states in the equations will be placed in the appropriate places from Eqs. (5.40) to (5.48). The interesting observation from

weight derivatives is that there is no recursive manner of derivatives like Eqs. (5.9) - (5.11) of RNN. This is the most significant property of LSTM over RNN. The constituent partial derivatives $\partial e_t/\partial U$ and $\partial e_t/\partial W$ now have contributions from the multiple instances of each weight U and W and depend on the inputs and hidden unit activations at previous time steps. The errors now have to be backpropagated through time as well as through the network.

5.5.1 Dimension of Weights in LSTM

In LSTM, input sequence collides with previous (i.e., existing) hidden and memory cell state through different weight sets and produces new hidden and memory cell state. The input dimension depends on input sequence length. Dimension of C_t and h_t are same and user defined. In general, the dimension of input sequence and C_t/ h_t are different. Therefore, dimension of weights is maintained to accomplish proper operation. Suppose input dimension is D and dimension of C_t/ h_t is H. Then weight dimension of W_f, W_c, W_i, and W_o is H×H. On the other hand, dimension of weights with input U_f, U_c, U_i, and U_o is D×H. Moreover, batch wise operation (i.e., perform operation on a set of patterns together) requires an additional dimension for batch size (i.e., number of patterns in a batch). For batch size B the third dimension weights will be B. Thus, dimension of W_f, W_c, W_i, and W_o will be $H\times H\times B$; dimension of U_f, U_c, U_i, and U_o is $D\times H\times B$. In such a case, dimension of input will be $D\times B$; and dimension of C_t and h_t will be $H\times B$. Fig. 5.13 shows the LSTM cell with dimension of weights and states for batch size B.

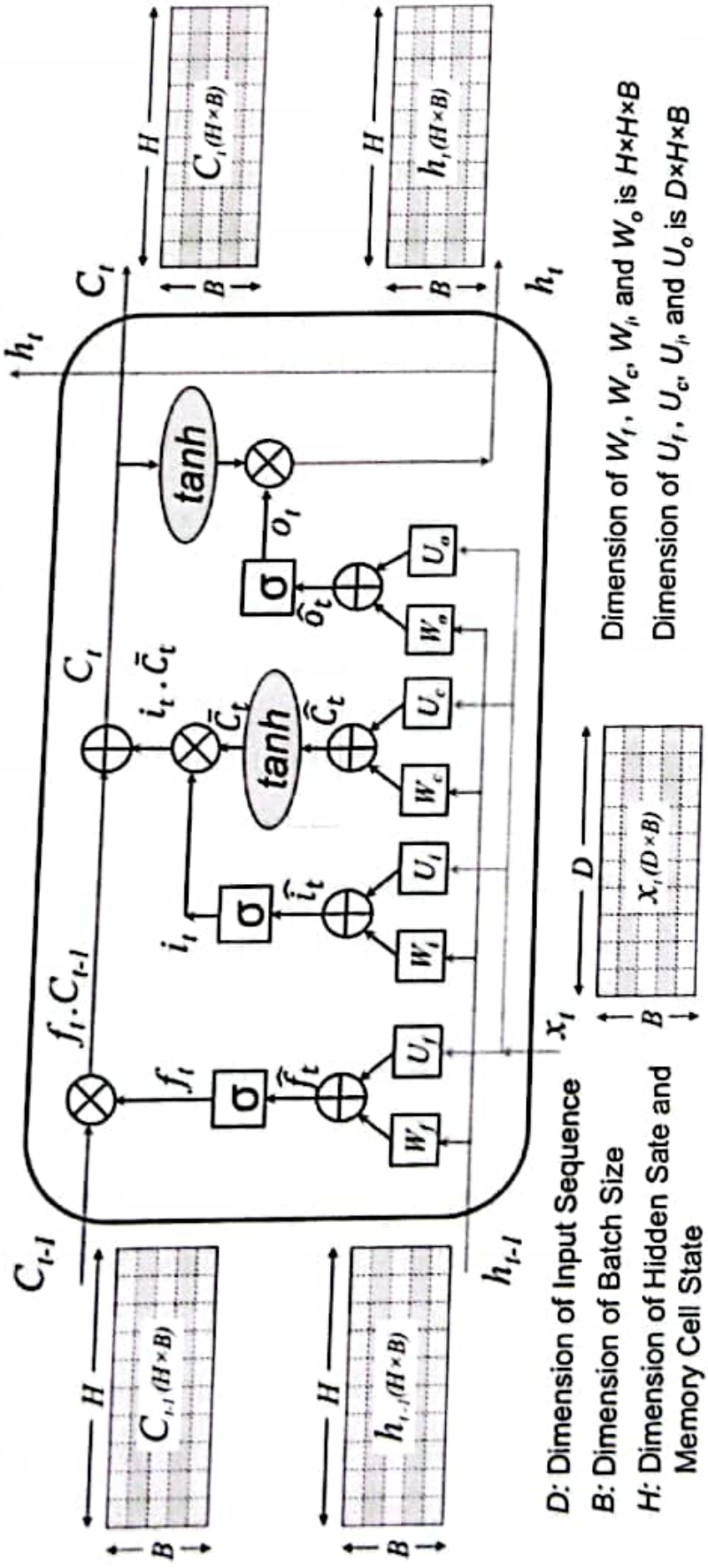


Figure 5.13: Weight dimension demonstration in LSTM with batch wise operation.

5.6 Details of a Sample LSTM Network and Training

LSTM is suitable to handle sequential data. This section demonstrates sequence prediction through processing some sample quotes of William Shakespeare. Moreover, handwritten numeral recognition is also presented using LSTM to harmonize with previous chapters and with motivation that LSTM is also useful for classification task.

5.6.1 Sequence Prediction with LSTM

A sample of quotes of William Shakespeare has been studied for sequence prediction in recent times. First few lines of the quotes are presented in Fig. 5.13. Total 66 characters are used in quotes to build words. Among characters 52 characters are for all 26 letters in both lower case and upper case. Two characters are reserved to identify START and END of a sequence respectively. Rest 12 characters include symbols, numeral and punctuations used in the quotes such as '!' and '$'. A dictionary is maintained to identify each character as numeric index as LSTM process numeric value only. Table 5.1 shows the dictionary. According to dictionary "`First Citizen:`" is "0 19 46 57 56 59 2 14 46 59 46 64 42 53 11". Here 0 indicates start of sequence and 2 for space. For training, the target sequence for the input sequence is "19 46 57 56 59 2 14 46 59 46 64 42 53 11 65" in which a particular character index is the target for the corresponding input character index shifting input index shifting one position. For the last input character target 65 indicates the end of sequence. For better understanding Table 5.2 shows transformation in tabular form. In