We are IntechOpen,
the world's leading publisher of
Open Access books
Built by scientists, for scientists

6,900
Open access books available

185,000
International authors and editors

200M
Downloads

154
Countries delivered to

Our authors are among the

TOP 1%
most cited scientists

12.2%
Contributors from top 500 universities

BOOK
CITATION
INDEX
CLARIVATE ANALYTICS
INDEXED

WEB OF SCIENCE™

Selection of our books indexed in the Book Citation Index
in Web of Science™ Core Collection (BKCI)

Interested in publishing with us?
Contact book.department@intechopen.com

Chapter

Deep Learning for Subtyping and Prediction of Diseases: Long-Short Term Memory

Hayrettin Okut

Abstract

The long short-term memory neural network (LSTM) is a type of recurrent neural network (RNN). During the training of RNN architecture, sequential information is used and travels through the neural network from input vector to the output neurons, while the error is calculated and propagated back through the network to update the network parameters. Information in these networks incorporates loops into the hidden layer. Loops allow information to flow multi-directionally so that the hidden state signifies past information held at a given time step. Consequently, the output is dependent on the previous predictions which are already known. However, RNNs have limited capacity to bridge more than a certain number of steps. Mainly this is due to the vanishing of gradients which causes the predictions to capture the short-term dependencies as information from earlier steps decays. As more layers in RNN containing activation functions are added, the gradient of the loss function approaches zero. The LSTM neural networks (LSTM-ANNs) enable learning long-term dependencies. LSTM introduces a memory unit and gate mechanism to enable capture of the long dependencies in a sequence. Therefore, LSTM networks can selectively remember or forget information and are capable of learn thousands timesteps by structures called cell states and three gates.

Keywords: deep learning, recurrent neural networks, long-short term memory

1. Introduction

Artificial neural networks (ANNs) are a type of the computing system that mimics and simulates the function of the human brain to analyze and process the complex data in an adaptive approach. They are capable of implementing massively parallel computations for mapping, function approximation, classification, and pattern recognition processing that require less formal statistical training. Moreover, ANNs have the ability to identify very high complex nonlinear relationships between outcome (dependent) and predictor (independent) variables using multiple training algorithms [1, 2]. Generally speaking, in terms of network architectures, ANNs tend be classified into two different classes, feedforward and recurrent ANNs, each may have several subclasses.

Feedforward is a widely used ANN paradigm for, classification, function approximation, mapping and pattern recognition. Each layer is fully connected to neurons in another layer with no connection between neurons in the same layer.

As the name suggests, information is fed in a forward direction from the input to the output layer through one or more hidden layers. MLP feed forward with one hidden layer can virtually predict any linear or non-linear model to any degree of accuracy, assuming that you have a appropriate number of neurons in hidden layer and an appropriate amount of data. Adding more neurons in the hidden layers to an ANN architecture gives the model the flexibility of fitting extremely complex nonlinear functions. This also holds true in classification modeling in approximating any nonlinear decision boundary with great accuracy [2].

Recurrent neural networks (RNNs) emerged as an operative and scalable ANN model for several learning problems associated with sequential data. Information in these networks incorporate loops into the hidden layer. These loops allow information to flow multi-directionally so that the hidden state signifies past information held at a given time step. Thus, these network types have an infinite dynamic response to sequential data. Many applications, such as Apple's Siri and Google's voice search, use RNN.

The most popular way to train a neural network is by backpropagation (BP). This method can be used with either feedforward or recurrent networks. BP involves working backward through each timestep to calculate prediction errors and estimate a gradient, which in turn is used to update the weights in the network. For example, to enable the long sequences found in RNNs, multiple timesteps are conducted that unrolls the network, adds new layers, and recalculates the prediction error, resulting in a very deep network.

However, standard and deep RNN neural networks may suffer from vanishing or exploding gradients problems. As more layers in RNN containing activation functions are added, the gradient of the loss function approaches zero. As more layers of activating functions are added, the gradient loss function may approach zero (vanish), leaving the functions unchanged. This stops further training and ends the procedure prematurely. As such, parameters capture only short-term dependencies, while information from earlier time steps may be forgotten. Thus, the model converges on a poor solution. Because error gradients can be unstable, the reverse issue, exploding gradients may occur. This causes errors to grow drastically within each time step (MATLAB, 2020b). Therefore, backpropagation may be limited by the number of timesteps.

Long Short-Term Memory (LSTM) networks address the issue of vanishing/exploding gradients and was first introduced by [3]. In addition to the hidden state in RNN, an LSTM block includes memory cells (that store previous information) and introduces a series of gates, called input, output, and forget gates. These gates allow for additional adjustments (to account for nonlinearity) and prevents errors from vanishing or exploding. The result is a more accurate predicted outcome, the solution does not stop prematurely, nor is previous information lost [4].

Practical applications of LSTM have been published in medical journals. For example, studies [5–14] used different variants of RNN for classification and prediction purposes from medical records. Important, LSTM does not have any assumption about elapsed time measures can be utilized to subtype patients or diseases. In one such study, LSTM was used to make risk predictions of disease progression for patients with Parkinson's by leveraging longitudinal medical records with irregular time intervals [15].

The purpose of this chapter is to introduce Long Short-Term Memory (LSTM) networks. This chapter begins with introduction of multilayer feedforward architectures. The core characteristic of an RNN and vanishing of the gradient will be explained briefly. Next, the LSTM neural network, optimization of network parameters, and the methodology for avoiding the vanishing gradient problem will be covered, respectively. The chapter ends with a MATLAB example for LSTM.

2. Artificial neural networks and multilayer neural network

Artificial Neural Networks (ANNs) are powerful computing techniques that mimic functions of the human brain to solve complex problems arising from big and messy data. As a machine learning method, ANNs can act as universal approximators of complex functions capable of capturing nonlinear relationships between inputs and outcomes. They adaptively learn functional structures by simultaneously utilizing a series of nonlinear and linear activation functions. ANNs offer several advantages. They require less formal statistical training, have the ability to detect all possible interactions between input variables, and include training algorithms adapted from backpropagation algorithms to improve the predictive ability of the model [2].

Feed forward multilayer perceptron (**Figure 1**) is the most used in ANN architectures. Uses include function approximation, classification, and pattern recognition. Similar to information processing within the human brain, connections are formed from successive layers. They are fully connected because neurons in each layer are connected to the neurons from the previous and the subsequent layer through adaptable synaptic network parameters. The first layer of multi-layer ANN is called the input layer (or left-most layer) that accepts the training data from sources external to the network. The last layer (or rightmost layer) is called the output layer that contains output units of the network. Depending on prediction or classification, the number of neurons in the output layer may consist of one or more neurons. The layer(s) between input and output layers are called hidden layer(s). Depending on the architecture multiple hidden layers can be placed between input and output layers.

Training occurs at the neuronal level in the hidden and output layers by updating synaptic strengths, eliminating some, and building new synapses. The central idea is to distribute the error function across the hidden layers, corresponding to their effect on the output. **Figure 1** demonstrates the architecture
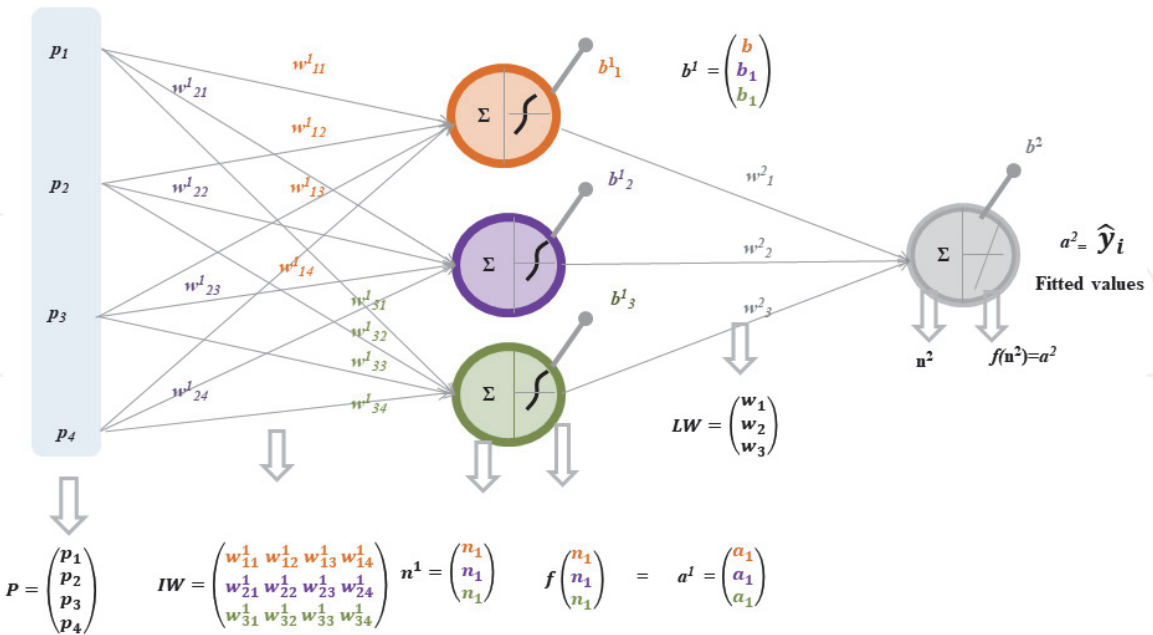

Figure 1.
*(adapted from Okut, 2016). Artificial neural network design with 4 inputs (p_i). Each input is connected to up to 3 neurons via coefficients $w_{kj}^{(l)}$ (l denotes layer; j denotes neuron; k denotes input variable). Each hidden and output neuron has a bias parameter b_j^l. Here P = inputs, **IW** = weights from input to hidden layer (12 weights), **LW** = weights from hidden to output layer (3 weights), b^1 = Hidden layer biases (3 biases), b^2 = Output layer biases (1 bias), n^1 = **IWP** + b^1 is the weighted summation of the first layer, a^1 = f(n^1) is output of hidden layer, n^2 = **LW**a^1 + b 2 is weighted summation of the second layer and $\hat{t}$ = a^2 = f(n^2) is the predicted value of the network. The total number of parameters for this ANN is 12 + 3 + 3 + 1 = 19.*

of a simple feedforward MLP where P identifies the input layer, next one or more hidden layers, which is followed by the output layer containing the fitted values. The feed forward MLP networks is evaluated in two stages. First, in the feedforward stage information comes from the left and each unit evaluates its activation function f. The results (output) are transmitted to the units connected to the right. The second stage involves the backpropagation (BP) step and is used for training the neural network using gradient descent algorithm in which the network parameters are moved along the negative of the gradient of the performance function The process consists of running the whole network backward and adjusting the weights (and error) in the hidden layer. The feedforward and backward steps are repeated several times, called epochs. The *algorithm stops when the value of the loss (error) function has become sufficiently small*.

3. Recurrent neural networks

Because the two stages, outlined in **Figure 1** above, are used in all neural networks, we can extend MLP feedforward neural networks for use in sequential data or time series data. To do so, we must address time sequences. Recurrent neural networks (RNNs) address this issue by conducting multiple time steps that unrolls the network, adds new layers, and recalculates the prediction error, resulting in a very deep network. First, the connections between nodes in the hidden layer(s) form a directed graph along a temporal sequence allowing information to persist. Through such a mechanism, the concept of time creates the RNNs memory. Here, the architecture receives information from multiple previous layers of the network.

Figure 2 outlines the hidden layer for RNN and demonstrates the nonlinear function of the previous layers and the current input (p). Here, the hyperbolic tangent activation function is used to generate the hidden state. The model has memory since the bias term is based on the "past". As a consequence, the outputs from the previous step are fed as input to the current step. Another way to think about RNNs is that a recurrent neural network has multiple copies of the same network, each passing a message to a successor (**Figure 3**). Thus, the output value of the last time point is transmitted back to the neural network, so that the parameter estimation (weight calculation) of each time point is related to the content of the previous time point.

3.1 Training recurrent neural networks

Similar to feedforward MLP networks, RNNs have two stages, a forward and a backward stage. Each works together during the training of the network. However, structures and calculation patterns differ. Let us first consider the forward pass.

Stage 1: Forward pass.

The forward pass will be summarized into 5 steps:

1. Summation step. In this step two different source of information are combined before nonlinear activation function will be take place. The sources are the values of weighted input $\left(W^{(p)}p_t\right)$ and weighted previous hidden state with bias $\left(h_{t-1}W^{(h)} + b^{(h)}\right)$. Here, p_t and $W^{(p)}$ are input vector and the input weight matrix, h_{t-1} is value of previous hidden state, $W^{(h)}$ is weight matrix of hidden state pertains the previous hidden state with the current one and $b^{(h)}$. Is bias. Since the previous hidden state and current input are measured as

vectors, each element in the vector is placed in a different orthogonal dimension

$$a_h(t) = \left(W^{(p)} p_t + h_{t-1} W^{(h)} + b^{(h)} \right) \tag{1}$$
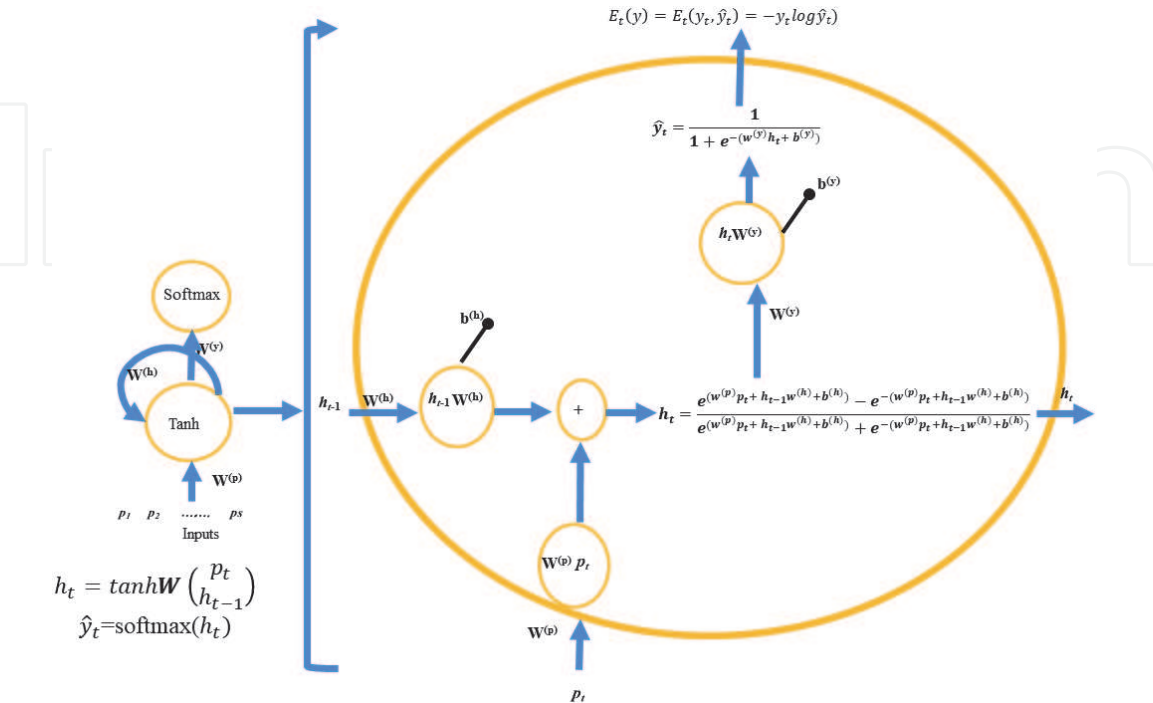

Figure 2.
A typical RNN that has a hyperbolic tangent activation function $\left(\frac{e^{(x)} - e^{-(x)}}{e^{(x)} + e^{-(x)}}\right)$ to generate the hidden state. Because of the hidden state RNNs have a "memory" that information has been calculated so far is captured. The information in hidden state passed further to a second activation function $\left(\frac{1}{1+e^{-(x)}}\right)$ to generate the predicted (output) values. In RNNs, the weight ($\mathbf{W}$) calculation of each time point of the network model is related to the content of the previous time point. We can process a sequence of vectors of inputs ($\mathbf{p}$) by applying a recurrence formula at every time step.
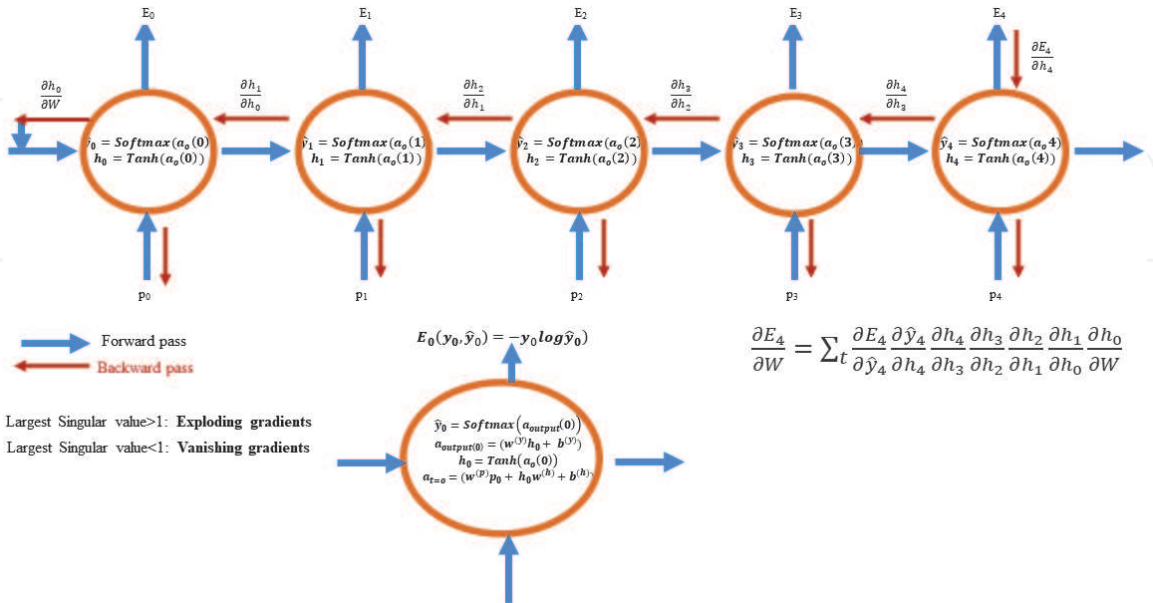

Figure 3.
An unrolled RNN with a hidden state carries pertinent information from one input item in the series to others. The blue and red arrows in the figure are indicating the forward and the backward pass of the network, respectively. With backward pass, we sum up the contributions of each time step to the gradient. In other words, because W is used in every step up to the output, we need to backpropagate gradients from $t = 4$ through the network all the way to $t = 0$.

The weight of the previous hidden state and current input are placed in a trainable weight matrix. Element-wise multiplication of the previous hidden state vector with the hidden state weights $(h_{t-1}W^{(h)})$ and element wise multiplication of the current input vector with the current input weights $(W^{(p)}p_t)$ produces the parameterized of state vector and input vector.

2. Hyperbolic tangent activation function is applied to the summed of the two parameterized vectors $\left(W^{(p)}p_t + h_{t-1}W^{(h)} + b^{(h)}\right)$ to push the output between -1 and 1 (**Figure 2**).

$$f(a_h(t)) = h_t = \frac{e^{\left(W^{(p)}p_t + h_{t-1}W^{(h)} + b^{(h)}\right)} - e^{-\left(W^{(p)}p_t + h_{t-1}W + b^{(h)}\right)}}{e^{\left(W^{(p)}p_t + h_{t-1}W^{(h)} + b^{(h)}\right)} + e^{-\left(W^{(p)}p_t + h_{t-1}W + b^{(h)}\right)}} \tag{2}$$

3. The network input to the output unit at time t with element-wise multiplication of output weights and with updated (current) hidden state $\left(h_t W^{(y)}\right)$.

   Therefore, the value before a softmax activation function takes place is $a_o(t) = h_t W^{(y)} + b^{(y)}$. Here $W^{(y)}$ and $b^{(y)}$ are the weight and bias of the output layer.

4. The output of the network at time t is calculated (the activation function applied to the output layer depends on the type of target (dependent) variable and the values coming from the hidden units. Again, a second activation function (mostly sigmoid) is applied to the value generated by the hidden node. The predicted value of a RNN block with sigmoid:

$$\hat{y}_t = \frac{1}{1 + e^{\left(W^{(y)}h_t + b^{(y)}\right)}} \tag{3}$$

   During the training of the forward pass of the RNN, the network outputs predicted value $(\hat{y}_i, i = t - 1, t, t + 1, \ldots, t + s)$ at each time step. We can image the unfold (unroll) of RNN given in **Figure 3**. That is, for each time step, an RNN can be imaged as multiple copies of the same network for the complete sequence. For example, if the sequence is a sentence of four words as; "*I have kidney problem*" then the RNN would be unrolled into a 4-layer neural network, one layer for each word. The output given in (3) is used to train the network using gradient descent after calculation of error in (4).

5. Then the error (loss function, cost function) at each time step is calculated to start the "*backward pass*":

$$E_t(y_t, \hat{y}_t) = -y_t \log \hat{y}_t) \tag{4}$$

Here y and $\hat{y}_t$ are actual and predicted outcomes, respectively. After calculation of the error at each time step, this calculated error is injected backwards into the network to update the network weights at each epoch (iteration). As there are many training algorithms based on some modification of standard backpropagation, the chosen error measure can be different and depends on the selected algorithm. For example, the error $E_t(y) = E_t(y_t, \hat{y}_t)$ given in (4), has an additional term, $E_t(w)$, in the Bayesian regularized neural networks (BRANN) training algorithm that penalizes large weights in anticipation of achieving smoother mapping. Both $E_t(y)$, $E_t(w)$, have a coefficient as β and α, respectively (also referred to as regularization parameters or hyper-parameters) that need to be estimated adaptively [1, 16].

Stage 2: Backward Pass.

After the forward pass of RNN, the calculated error (loss function, cost function) at each time step is injected backwards into the network to update the network weights at each iteration. The idea of RNN unfolding in **Figure 3** takes place the bigger part in the way RNNs are implemented for the backward pass. Like standard backpropagation in feed forward MLP, the backward pass consists of a repeated application of the chain rule. For this reason, the type of backpropagation algorithm used for an RNN to update the network parameters is called backpropagation through time (BPTT). In BPTT, the RNN network is unfolded in time to construct a feed forward MLP neural network. Then, the generalized delta rule is applied to update the weights $W^{(p)}$, $W^{(h)}$ and $W^{(y)}$ and biases $b^{(h)}$ and $b^{(y)}$. Remember, the goal with backpropagation is minimizing the gradients of error with respect to the network parameter space ($W^{(p)}$, $W^{(h)}$ and $W^{(y)}$ and biases $b^{(h)}$ and $b^{(y)}$) and then updates the parameters using Stochastic Gradient Descent. The following equation is used to update the parameters for minimizing the error function:

$$W_t = W_{t-1} - a \frac{\partial E}{\partial W}.$$

Here, a is learning rate and $\frac{\partial E}{\partial W}$ is the derivative of the error function with respect to parameters space. The same is applied to all weights and biases on the networks. The error each time step is

$$E_0(y, \hat{y}) = -y log \hat{y}$$

The total error is calculated by the summation of the error from all time steps as:

$$E = \sum_t - y_t log \hat{y}_t \tag{5}$$

The value of gradients $\frac{\partial E}{\partial W}$ at each time step is calculated as (the same rule is applied to all parameters on the network):

$$\frac{\partial E}{\partial W} = \sum_t \frac{\partial E_t}{\partial W} \tag{6}$$

To calculate the error gradient given in Eq. (6):

$$\frac{\partial E_t}{\partial W} = \frac{\partial E_t}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial h_t} \frac{\partial h_t}{\partial h_{t-1}} \frac{\partial h_{t-1}}{\partial h_{t-2}} \frac{\partial h_{t-2}}{\partial h_{t-3}} \cdots \cdots \cdots \cdots \frac{\partial h_0}{\partial W}$$

To calculate the overall error gradient, the chain rule of differentiation given in (7) is used.

$$\frac{\partial E}{\partial W} = \sum_t \frac{\partial E_t}{\partial \hat{y}_t} \frac{\partial \hat{y}_t}{\partial h_t} \frac{\partial h_t}{\partial h_{t-1}} \frac{\partial h_{t-1}}{\partial h_{t-2}} \frac{\partial h_{t-2}}{\partial h_{t-3}} \cdots \cdots \cdots \cdots \frac{\partial h_0}{\partial W} \tag{7}$$

Then the network weights can be updated as follow:

$$W_{t+1}^{(h)} = W_t^{(h)} + a \frac{\partial E}{\partial W^{(h)}}$$

$$W_{t+1}^{(p)} = W_t^{(p)} + a \frac{\partial E}{\partial W^{(p)}}$$

$$W_{t+1}^{(y)} = W_t^{(y)} + a\frac{\partial E}{\partial W^{(y)}}$$

Note that, as given in (2), the current state $(h_t) = tanh\left(W^{(p)}p_t + h_{t-1}W^{(h)} + b^{(h)}\right)$ depends on the quantity of the previous state (h_{t-1}) and the other parameters. Therefore, the differentiation of h_t and h_j (here $j = 0, 1, ..., t-1$) given in (7) is a derivative of a hidden state that stores memory at time t.

The Jacobians of any time $\left(\frac{\partial h_j}{\partial h_{j-1}}\right)$ and for the entire time will be:

$$\frac{\partial h_j}{\partial h_{j-1}} = \frac{\partial h_t}{\partial h_{t-1}}\frac{\partial h_{t-1}}{\partial h_{t-2}}\frac{\partial h_{t-2}}{\partial h_{t-3}} \cdots \cdots \cdot \frac{\partial h_{j+1}}{\partial h_j} = \prod_{j=j+1}^{t}\frac{\partial h_j}{\partial h_{j-1}} \tag{8}$$

while the Jacobian matrix for hidden state is given by:

$$\frac{\partial h_j}{\partial h_{j-1}} = \left[\frac{\partial h_j}{\partial h_{j-1,1}}, \frac{\partial h_j}{\partial h_{j-1,2}} \cdots \frac{\partial h_j}{\partial h_{j-1,s}}\right] = \begin{bmatrix} \frac{\partial h_{j,1}}{\partial h_{j-1,1}} & \cdots & \frac{\partial h_{j,1}}{\partial h_{j-1,s}} \\ \vdots & \ddots & \vdots \\ \frac{\partial h_{j,s}}{\partial h_{j-1,1}} & \cdots & \frac{\partial h_{j,s}}{\partial h_{j-1,s}} \end{bmatrix} \tag{9}$$

Putting the Eqs. (7) and (8) together, we have the following relationship:

$$\frac{\partial E}{\partial W} = \sum_t \sum_j \frac{\partial E_t}{\partial y_t}\frac{\partial y_t}{\partial h_t}\left(\prod_{j=j+1}^{t}\frac{\partial h_j}{\partial h_{j-1}}\right)\frac{\partial h_j}{\partial W}. \tag{10}$$

In other words, because the network parameters are used in every step up to the output, we need to backpropagate gradients from last time step $(t = t)$ through the network all the way to $t = 0$. The Jacobians in (10), $\left(\frac{\partial h_j}{\partial h_{j-1}}\right)$, demonstrates the eigen decomposition given by $W^{(i)T}diag\left(f'(h_{j-1})\right)$, where the eigenvalues and eigenvectors are generated. Here the $W^{(i)T}$ is the transpose of the network parameters matrix. Consequently, if the largest eigenvalue is greater or smaller than 1, the RNN suffers from vanishing or exploding gradient problems (see **Figure 3**).

4. Long short-term memory

As mentioned before, the output from RNNs is dependent on its previous state or previous N time steps circumstances. Conventional RNN face difficulty in learning and maintaining long-range dependencies. Imagine the unfolding RNN given in **Figure 3**. Each time step requires a new copy of the network. With large RNNs, thousands, even millions of weights are needed to be updated. In other word, $\frac{\partial h_j}{\partial h_{j-1}}$ is a chain rule itself. For **Figure 3**, for example, the derivative of $\frac{\partial h_4}{\partial h_3} = \frac{\partial h_4}{\partial h_3}\frac{\partial h_3}{\partial h_2}\frac{\partial h_2}{\partial h_1}\frac{\partial h_1}{\partial h_0}$. Imagine an unrolling the RNN a thousand times, in which every activation of the neurons inside the network are replicated thousands of times. This means, especially for larger networks, that thousands or millions of weights are needed. As Jacobian matrix will play a role to update the weights, the values of the Jacobian matrix will range between $-1, 1$ if tanh activation function is applied to the $f(a_h(t)) = h_t = tanh\left(W^{(y)}h_t + b^{(y)}\right)$ given in (2). It can be easily imagined that the

derivatives of tanh (or sigmoid) activation function would be 0 at the end. Zero gradients drive other gradients in previous layers towards 0. Thus, with small values in the Jacobian matrix and multiple matrix multiplications (t-j, in particular) the gradient values will be shrunk exponentially fast, eventually vanishing completely after a few time steps. As a result, the RNN ends up not learning long-range dependencies. As in RNNs, the vanishing gradients problem will be an important issue for the deep feedforward MLP when multiple hidden layers (multiple neurons within each) are placed between input and output layers.

The long short-term memory networks (LSTMs) are a special type of RNN that can overcome the vanishing gradient problem and can learn long-term dependencies. LSTM introduces a memory unit and a gate mechanism to enable capture of the long dependencies in a sequence. The term "long short-term memory" originates from the following intuition. Simple RNN networks have long-term memory in the form of weights. The weights change gradually during the training of the network, encoding general knowledge about the training data. They also have short-term memory in the form of ephemeral activations, which flows from each node to successive nodes [17, 18].

4.1 The architecture of LSTM

The neural network architecture for an LSTM block given in **Figure 4** demonstrates that the LSTM network extends RNN's memory and can selectively remember or forget information by structures called cell states and three gates. Thus, in addition to a hidden state in RNN, an LSTM block typically has four more layers. These layers are called the cell state (C_t), an input gate (i_t), an output gate (O_t), and a forget gate (f_t). Each layer interacts with each other in a very special way to generate information from the training data.
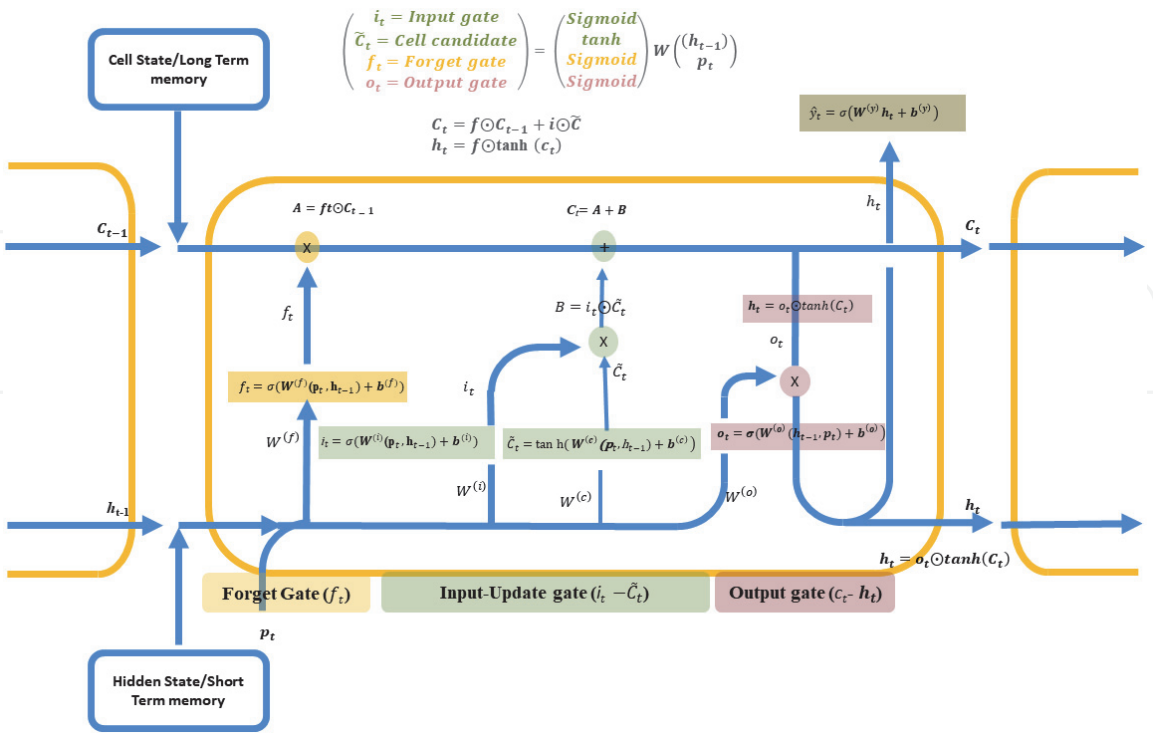

Figure 4.
Illustration of long short-term memory block structure. The operator "$\odot$" denotes the element-wise multiplication. The C_{t-1}, C_t, h_t and h_t are previous cell state, current cell state, current hidden state and previous hidden state, respectively. The f_t; i_t; o_t are the values of the forget, input and output gates, respectively. The $\tilde{C}_t$ is the candidate value for the cell state, $W^{(f)}$, $W^{(i)}$, $W^{(c)}$, $W^{(o)}$ are weight matrices consist of forget gate, input gate, cell state and output gate weights, and $b^{(f)}$, $b^{(i)}$, $b^{(c)}$, and $b^{(o)}$ are bias vectors associated with them.

A block diagram of LSTM at any timestamp is depicted in **Figure 4**. This block is a recurrently connected subnet that contains the same cell state and three gates structure. The p_t, h_{t-1}, and C_{t-1} correspond to the input of the current time step, the hidden output from the previous LSTM unit, and the cell state (memory) of the previous unit, respectively. The information from the previous LSTM unit is combined with current input to generate a newly predicted value. The LSTM block is mainly divided into three gates: forget (blue), input-update (green), and output (red). Each of these gates is connected to the cell state to provide the necessary information that flows from the current time step to the next.

A sigmoid activation function $\left(\frac{1}{1+e^{-(x)}}\right)$ is implemented in the forget gate. For the input and output gates, however, a combination of sigmoid and hyperbolic tangent-tanh-$\left(\frac{e^{(x)}-e^{-(x)}}{e^{(x)}+e^{-(x)}}\right)$ are used to provide the necessary information to the cell state. The information generated by the blocks flow through the cell state from one block to another as the chain of repeating components of the LSTM neural network holds. Details about cell state and each layer are given in different subtitles.

4.1.1 Cell state

As shown in the upper part of **Figures 4** and **5**, the cell state is the key to LSTMs and represents the memory of LSTM networks. The process for the cell state is very much like to a conveyor belt or production chain. The information about the parameters runs straight forward the entire chain, with only some linear interactions, such as multiplication and addition. The state of information depends on these interactions. If there are no interactions, the information will run along without changes. The LSTM block removes or adds information to the cell state through the gates, which allow optional information to cross [19].

4.1.2 Forget gate

The *Forget Gate* (f_t) decides the type of information that should be thrown away or kept from the cell state. This process is implemented by a sigmoid activation function. The sigmoid activation function outputs values between 0 and 1 coming from the weighted input $(W_f p_t)$, previous hidden state (h_{t-1}), and a bias (b_f). The forget gates (**Figure 6**) can be described by the equation given in (11). Here, σ is the
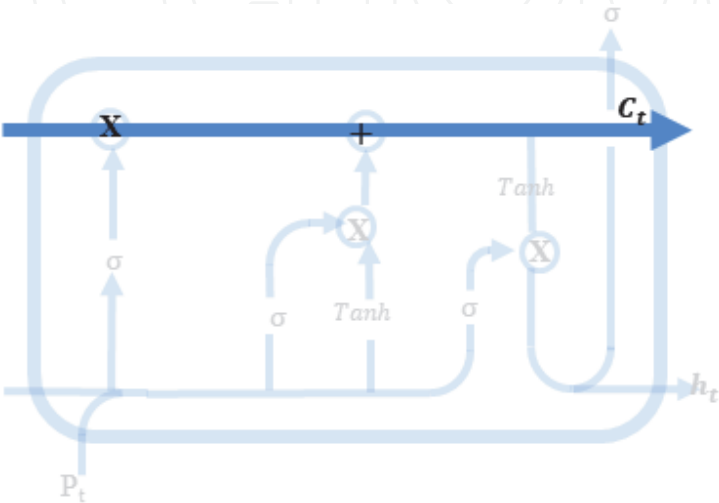

Figure 5.
The cell state, the horizontal line running through the top of the diagram of an LSTM.

sigmoid activation function, $W^{(f)}$ and $b^{(f)}$ are the weight matrix and bias vector, which will be learned from the input training data.

$$f_t = \sigma\left(W^{(f)}(p_t, h_{t-1}) + b^{(f)}\right) = \frac{1}{1 + e^{-\left(W^{(f)} \cdot (p_t, h_{t-1}) + b^{(f)}\right)}} \qquad (11)$$

The function takes the old output (h_{t-1}) at time $t-1$ and the current input (p_t) at time t for calculating the components that control the cell state and hidden state of the layer. The results are [0,1], where 1 represents "completely hold this" and 0 represents "completely throw this away" (**Figure 6**).

4.1.3 Input gate

The *Input Gate* (i_t) controls what new information will be added to the cell state from the current input. This gate also plays the role to protect the memory contents from perturbation by irrelevant input (**Figures 7** and **8**). A sigmoid activation function is used to generate the input values and converts information between 0 and 1. So, mathematically the input gate is:

$$i_t = \sigma\left(W^{(i)}(p_t, h_{t-1}) + b^{(i)}\right) = \frac{1}{1 + e^{-\left(W^{(i)} \cdot (p_t, h_{t-1}) + b^{(i)}\right)}} \qquad (12)$$

where $W^{(i)}$ and $b^{(i)}$ are the weight matrix and bias vector, p_t is the current input timestep index with the previous time step h_{t-1}. Similar to the forget gate, the parameters in the input gate will be learned from the input training data. At each time step, with the new information p_t, we can compute a candidate cell state.

Next, a vector of new candidate values, $\tilde{C}_t$, is created. The computation of the new candidate is similar to that of (11) and (12) but uses a hyperbolic tanh activation function with a value range of $(-1,1)$. This leads to the following Eq. (13) at time t.

$$\tilde{C}_t = \tanh\left(W^{(c)}(p_t, h_{t-1}) + b^{(c)}\right) = \frac{e^{\left(W^{(c)} \cdot (p_t, h_{t-1}) + b^{(c)}\right)} - e^{-\left(W^{(c)} \cdot (p_t, h_{t-1}) + b^{(c)}\right)}}{e^{\left(W^{(c)} \cdot (p_t, h_{t-1}) + b^{(c)}\right)} + e^{-\left(W^{(c)} \cdot (p_t, h_{t-1}) + b^{(c)}\right)}} \qquad (13)$$

In the next step, the values of the input state and cell candidate are combined to create and update the cell state as given in (14). The linear combination of the input gate and forget gate are used for updating the previous cell state (C_{t-1}) into current
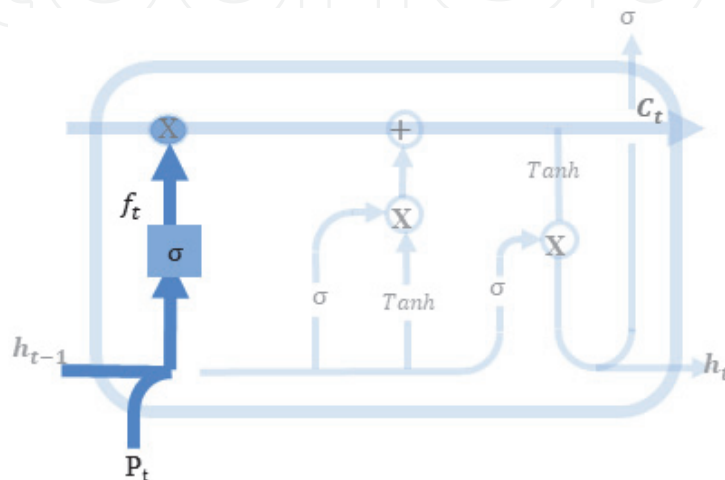

Figure 6.
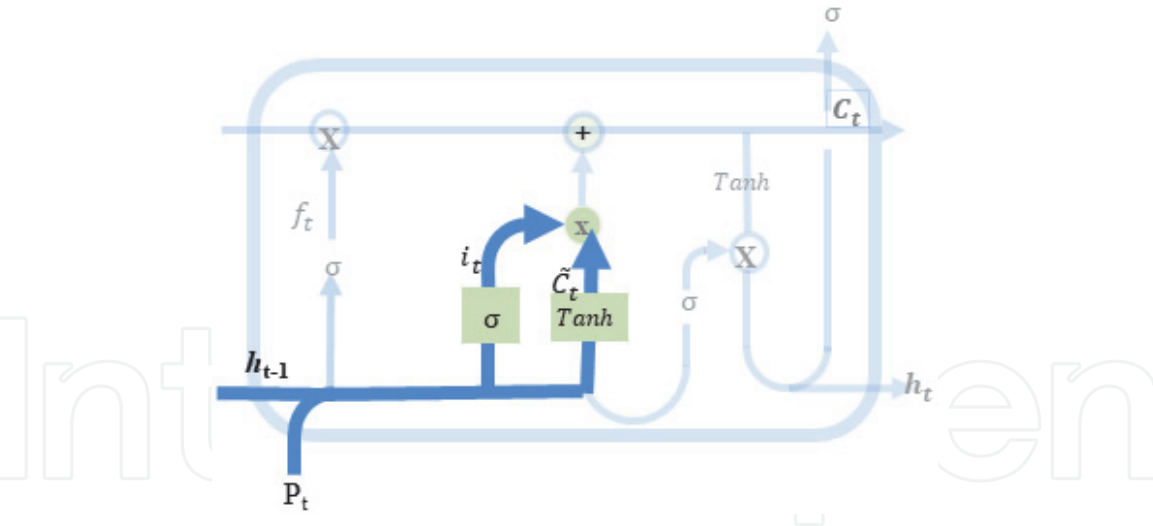*The forget gate controls what information to throw away from the memory.*

Figure 7.
The input-update gate decides what new information should be stored in the cell state, which has two parts: A sigmoid layer and a hyperbolic tangent (tanh) layer. The sigmoid layer is called the "input gate layer" because it decides which values should be updated. The tanh layer is a vector of new candidate values $\tilde{C}_t$ that could be added to the cell state.
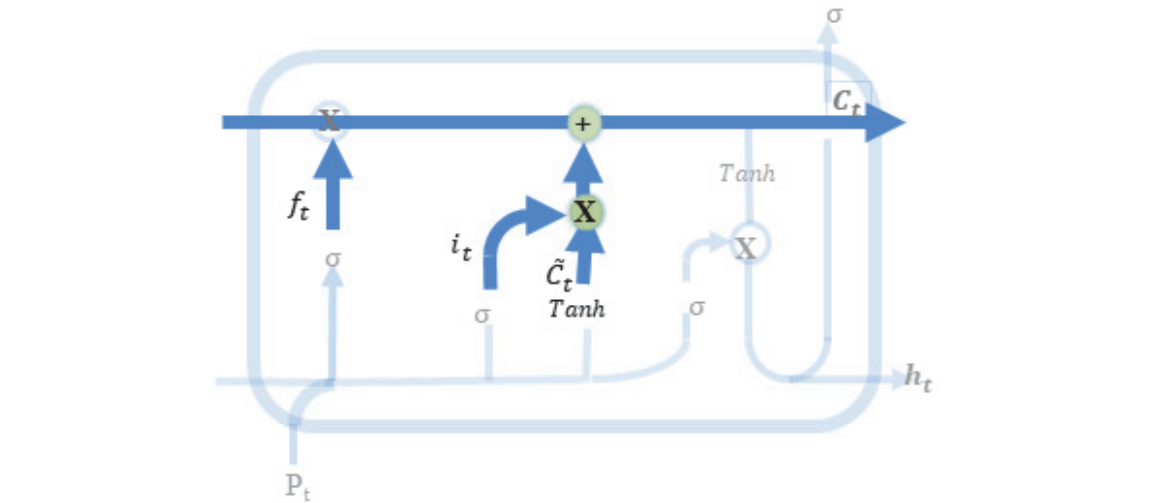

Figure 8.
Memory update is done using old memory via the forget gate and new memory via the input gate.

cell state (C_t). Once again, the input gate (i_t) governs how much new data should be taken into account via the candidate ($\tilde{C}_t$), while the forget gate (f_t) reports how much of the old memory cell content (C_{t-1}) should be retained. Using the same pointwise multiplication ($\odot$ =Hadamard product), we arrive at the following updated equation:

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t \qquad (14)$$

4.1.4 Output gate

The *Output Gate* (o_t) controls which information to reveal from the updated cell state (C_t) to the output in a single time step. In other words, the output gate determines what the value of the next hidden state should be in each time step. As depicted in **Figure 9**, the hidden state comprises information on previous inputs. Moreover, the calculated value of the hidden state for the given time step is used for
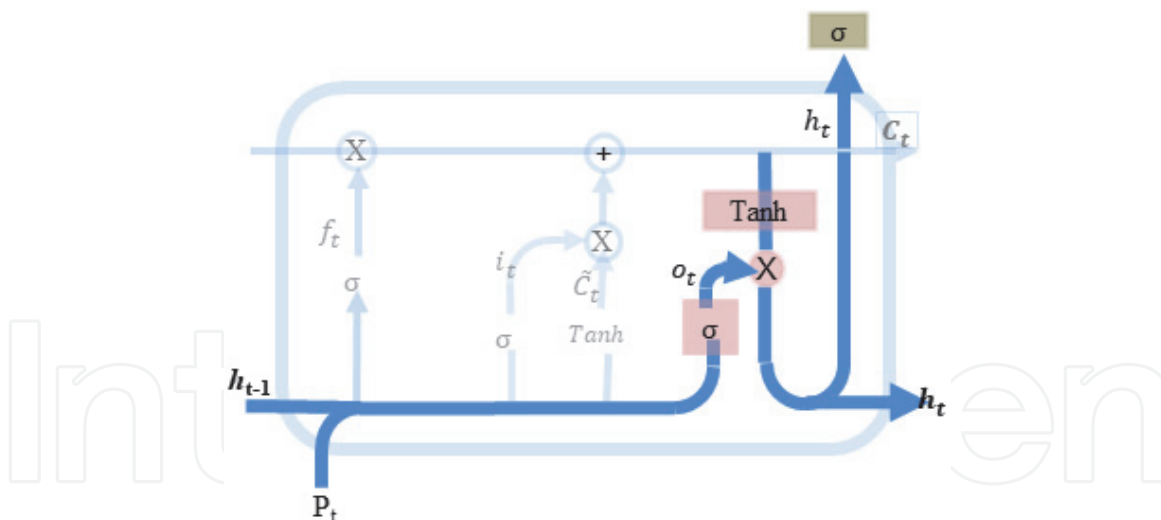
Figure 9.
The output state decides what information will be output using a sigmoid σ and tanh (to push the values to be between −1 and 1) layers.

the prediction ($\hat{y}_t = \text{softmax}(.)$). Here, softmax is a nonlinear activation function (sigmoid, hyperbolic tangent etc.).

$$o_t = \sigma\left(W^{(o)}\left(h_{t-1}, p_t\right) + b^{(o)}\right) = \frac{1}{1 + e^{-\left(W^{(o)}\left(h_{t-1}, p_t\right) + b^{(o)}\right)}} \tag{15}$$

$$h_t = o_t \odot \tanh\left(C_t\right) = o_t \bullet \frac{e^{\left(f_t \odot C_{t-1} + i_t \odot \tilde{C}_t\right)} - e^{-\left(f_t \odot C_{t-1} + i_t \odot \tilde{C}_t\right)}}{e^{\left(f_t \odot C_{t-1} + i_t \odot \tilde{C}_t\right)} + e^{-\left(f_t \odot C_{t-1} + i_t \odot \tilde{C}_t\right)}} \tag{16}$$

$$\hat{y}_t = \sigma\left(W^{(y)}h_t + b^{(y)}\right) = \frac{1}{1 + e^{-\left(W^{(y)}h_t + b^{(y)}\right)}} \tag{17}$$

First, the previous hidden state (h_{t-1}) is passed to the current input into a sigmoid function. Next newly updated cell state is generated with the tanh function [15, 18]. Finally, the tanh output is multiplied with the sigmoid output to determine what information the hidden state should carry (16). The final product of the output gate is an updated of the hidden state, and this is used for the prediction at time step t. Therefore, the aim of this gate is to separate the updated cell state (updated memory) from the hidden state. The updated cell state (C_t) contains a lot of information that is not necessarily required to be saved in the updated hidden state. However, this information is critical as the updated hidden state at each time is used in all gates of an LSTM block. Thus, the output gate does the assessment regarding what parts of the cell state (C_t) is presented in the hidden state (h_t). The new cell and new hidden states are then passed to the next time step (**Figure 9**).

Summary of forward pass

1. *Forget gate*: Controls what information to throw away and decides how much from the part should be remember. $f_t = \sigma\left(W^{(f)}(p_t, h_{t-1}) + b^{(f)}\right)$

2. *Input-Update Gate:* Controls information to add cell state from current input and decides how much should be added to the cell state $i_t = \sigma\left(W^{(i)}(p_t, h_{t-1}) + b^{(i)}\right), \tilde{C}_t = \tanh\left(W^{(c)}(p_t, h_{t-1}) + b^{(c)}\right)$

---

3. *Output gate:* Determines the part of the current cell state makes it *to the output*
$$o_t = \sigma\left(W^{(o)}(h_{t-1}, p_t) + b^{(o)}\right).$$

4. *Current cell state:* $C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t$

5. *Current hidden state:* $h_t = o_t \odot tanh(C_t) \Rightarrow h_t = LSTM((p_t, h_{t-1})$

6. *LSTM block prediction:* $\hat{y}_t = \sigma\left(W^{(y)}h_t + b^{(y)}\right)$

7. *Calculate the LSTM block error for the time step:* $E_t(y_t, \hat{y}_t) = -y_t \log \hat{y}_t)$

4.2 Backward pass

Like the RNN networks, an LSTM network generates an output $(\hat{y}_t)$ at each time step that is used to train the network via gradient descent (**Figure 10**). During the backward pass, the network parameters are updated at each epoch (iteration). The only fundamental difference between the back-propagation algorithms of the RNN and LSTM networks is a minor modification of the algorithm. Here, the calculated error term at each time step is $E_t = -y_t \log \hat{y}_t$. As in RNN, the total error is calculated by the summation of error from all time steps $E = \sum_t -y_t \log \hat{y}_t$.

Similarly, the value of gradients $\frac{\partial E}{\partial W}$ at each time step is calculated and then the summation of the gradients at each time steps $\frac{\partial E}{\partial W} = \sum_t \frac{\partial E_t}{\partial W}$ is obtained. Remember, the predicted value, $\hat{y}_t$, is a function of the hidden state $(\hat{y}_t = \sigma\left(W^{(y)}h_t + b^{(y)}\right))$ and the hidden state (h_t) is a function of the cell state $(h_t = o_t \odot tanh(C_t))$. These both are subjected in the chain rule. Hence, the derivatives of individual error terms with respect the network parameter:

$$\frac{\partial E_t}{\partial W} = \frac{\partial E_t}{\partial \hat{y}_t}\frac{\partial \hat{y}_t}{\partial h_t}\frac{\partial h_t}{\partial c_t}\frac{\partial c_t}{\partial c_{t-1}}\frac{\partial c_{t-1}}{\partial c_{t-2}}\frac{\partial c_{t-2}}{\partial c_{t-3}} \dots \dots \dots \dots \cdot \frac{\partial c_0}{\partial W} \tag{18}$$

and for the overall error gradient using the chain rule of differentiation is:

$$\frac{\partial E}{\partial W} = \sum_t \frac{\partial E_t}{\partial \hat{y}_t}\frac{\partial \hat{y}_t}{\partial h_t}\frac{\partial h_t}{\partial c_t}\frac{\partial c_t}{\partial c_{t-1}}\frac{\partial c_{t-1}}{\partial c_{t-2}}\frac{\partial c_{t-2}}{\partial c_{t-3}} \dots \dots \dots \dots \cdot \frac{\partial c_0}{\partial W} \tag{19}$$

As Eq. (19) illustrates, the gradient involves the chain rule of ∂c_t in an LSTM training using the backpropagation algorithm, while the gradient equation involves a chain rule of ∂h_t for a basic RNN. Therefore, the Jacobian matrix for cell state for an LSTM is [20]:

$$\left[\frac{\partial c_j}{\partial c_{j-1,1}}, \frac{\partial c_j}{\partial c_{j-1,2}} \dots \frac{\partial c_j}{\partial c_{j-1,s}}\right] = \begin{bmatrix} \frac{\partial c_{j,1}}{\partial c_{j-1,1}} & \dots & \frac{\partial c_{j,1}}{\partial c_{j-1,s}} \\ \vdots & \ddots & \vdots \\ \frac{\partial c_{j,s}}{\partial c_{j-1,1}} & \dots & \frac{\partial c_{j,s}}{\partial c_{j-1,s}} \end{bmatrix} \tag{20}$$

The problem of gradient vanishing

Recall the Eq. (14) for cell state is $C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t$. When we consider Eq. (19), the value of the gradients in the LSTM is controlled by the chain of
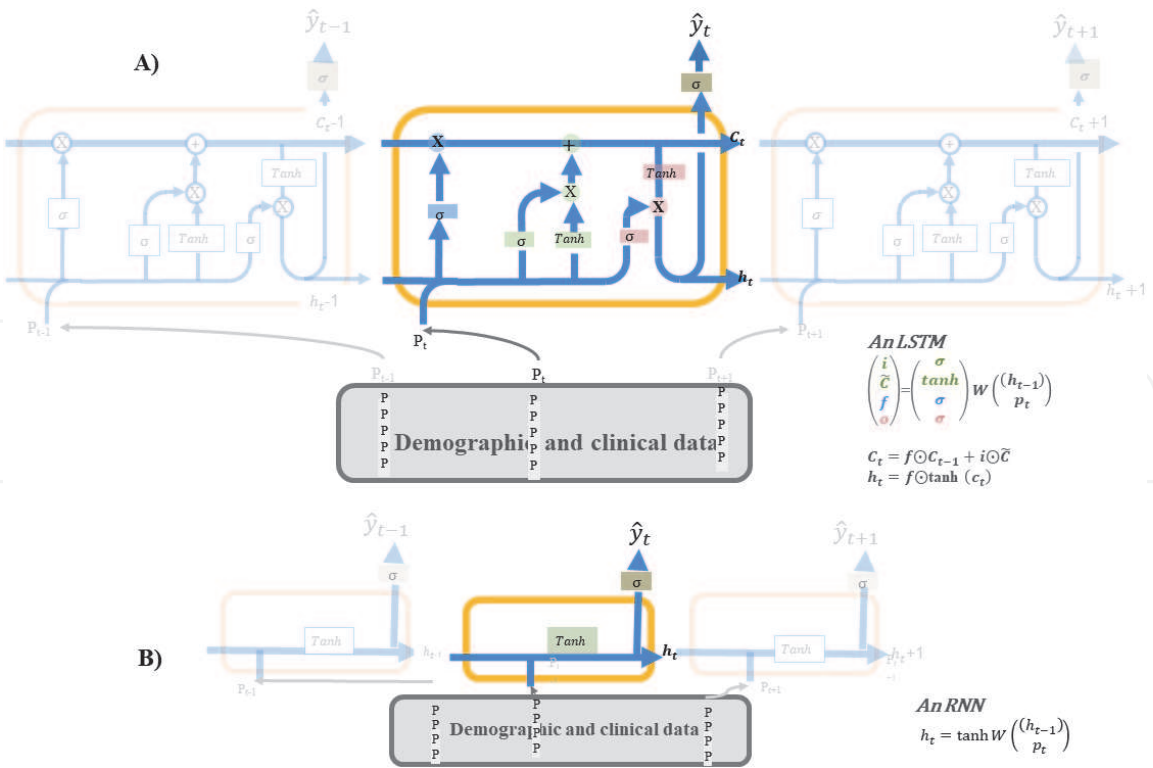
Figure 10.
*Illustration of the (A) an LSTM unit from 3-time steps with input data (demographic and clinical data). LSTM network takes inputs from to the current time step to update the hidden state and (**LSTM**$((p_t, h_{t-1}))$) with relevant information. The "x" in the circles denote point-wise operators, σ and tanh are sigmoid ($\left(\frac{1}{1+e^{-(x)}}\right)$, generates between 0 and 1) and hyperbolic tangent ($\left(\frac{e^{(x)}-e^{-(x)}}{e^{(x)}+e^{-(x)}}\right)$, generates between −1 and 1) activation functions. (B) an RNN with 3-time steps. It has only a tangent, $\left(\frac{e^{(x)}-e^{-(x)}}{e^{(x)}+e^{-(x)}}\right)$, activation function in the block.*

derivatives starting from the part $\frac{\partial c_t}{\partial c_{t-1}}$. Expanding this value using the expression for $C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t$.

$$\frac{\partial c_t}{\partial c_{t-1}} = \frac{\partial(f_t \odot C_{t-1} + i_t \odot \tilde{C}_t)}{\partial(f_{t-1} \odot C_{t-2} + i_{t-1} \odot \tilde{C}_{t-1})}$$
$$= \frac{\partial c_t}{\partial f} \frac{\partial f}{\partial h_{t-1}} \frac{\partial h_{t-1}}{\partial c_{t-1}} + \frac{\partial c_t}{\partial i} \frac{\partial i}{\partial h_{t-1}} \frac{\partial h_{t-1}}{\partial c_{t-1}} + \frac{\partial c_t}{\partial \tilde{C}_t} \frac{\partial \tilde{C}_t}{\partial h_{t-1}} \frac{\partial h_{t-1}}{\partial c_{t-1}} + \frac{\partial c_t}{\partial c_{t-1}} \qquad (21)$$

Note the term $\frac{\partial c_t}{\partial c_{t-1}}$ does not have a fixed pattern and can yield any positive value in the LSTM, while the $\frac{\partial c_t}{\partial c_{t-1}}$ term in the standard RNN can yield values greater than 1 or less than 1 after certain time steps. Thus, for an LSTM, the term will not converge to 0 or diverge completely, even for an infinite number of time steps. If the gradient starts converging towards zero, the weights of the gates are adjusted to bring it closer to 1.

4.3 Other type of LSTMs

Several modifications to original LSTM architecture have been recommended over the years. Surprisingly, the original continues to outperform, and has similar predictive ability compared with variants of LSTM over 20 years.

4.3.1 Peephole connections

This is a type of LSTM by adding "*peephole connections*" to the standard LSTM network. The theme stands for peephole connections needed to capture information

inherent to time lags. In other words, with *peephole connections* the information conveyed by time intervals between sub-patterns of sequences is included to the network recurrent. Thus, *peephole connections* concatenate the previous cell state (C_{t-1}) information to the forget, input and output gates. That is, the expression of these gates with peephole connection would be:

$$
\begin{aligned}
f_t &= \sigma\Big(W^{(f)}\big(p_t, h_{t-1}, C_{t-1}\big) + b^{(f)}\Big) \\
i_t &= \sigma\Big(W^{(i)}\big(p_t, h_{t-1}, C_{t-1}\big) + b^{(i)}\Big) \\
o_t &= \sigma\Big(W^{(o)}\big(h_{t-1}, p_t\big) + b^{(o)}\Big)
\end{aligned}
\tag{22}
$$

This configuration was offered to improve the predictive ability of LSTMs to count and time distances between rare events [21].

4.3.2 Gated recurrent units

Gated recurrent units (GRU) is a simplified version of the standard LSTM designed in a manner to have more persistent memory in order to make it easier for RNNs. A GRU is called gated RNN and introduced by [22]. In a GRU, forget and input gates are merged into a single gate named "update gate". Moreover, the cell state and hidden state are also merged. Therefore, the GRU has fewer parameters and has been shown to outperform LSTM on some tasks to capture long-term dependencies.

4.3.3 Multiplicative LSTMs (mLSTMs)

This configuration of LSTM was introduced by Krause et al., [23]. The architecture is for sequence modeling combines LSTM and multiplicative RNN architectures. mLSTM is characterized by its ability to have different recurrent transition functions for each possible input, which makes it more expressive for autoregressive density estimation. Krause et al. concluded that mLSTM outperforms standard LSTM and its deep variants for a range of character-level language modeling tasks.

4.3.4 LSTM with attention

The core idea behind the LSTM with Attention frees the encoder-decoder architecture from the fixed-length internal illustration. This is one of the most transformative innovations in sequence to uncover the mobility regularities in the hidden node of LSTM. The LSTM with attention was introduced by Wu et al., [24] for Bridging the Gap between Human and Machine Translation in Google's Neural Machine Translation System. The LSTM with attraction consists of a deep LSTM network with 8 encoder and 8 decoder layers using attention and residual connections. Most likely, this type of LSTM continues to power Google Translate to this day.

4.4 Examples

Two examples using MATLAB for LSTM will be given for this particular chapter.

Example **1** This example shows how to forecast time series data for COVID19 in the USA using a long short-term memory (LSTM) network. The variable used in

training data is the rate for the number of positive/number of tests for each day between 01/22/2020–2112/22/2020. Data set was taken from publicly available h ttps://covidtracking.com/data/national. web site and data are updated each day between about 6 pm and 7:30 pm Eastern Time Zone. The initiative relies upon publicly available data from multiple sources. States in the USA are not consistent in how and when they release and update their data, and some may even retroactively change the numbers they report. This can affect the predictions presented in these data visualizations (**Figure 11a-d**). The steps for example 1 are summarized in the **Table 1** (MATLAB 2020b) and results are illustrated in **Figure 11a-d.** LSTM network was trained on the first 90% of the sequence and tested on the last 10%. Therefore, results reveal predicting the positive last 38 days.

This example trains an LSTM network to forecast the number of positively tested given the number of cases in previous days. The training data contains a single time series, with time steps corresponding to days and values corresponding to the number of cases. To make predictions on a new sequence, reset the network state using the "*resetState*" command in MATLAB. Resetting the network state prevents previous predictions from affecting the predictions on the new data. Reset the network state, and then initialize the network state by predicting on the training data (MATLAB, 2020b). The solid line with red color in **Figure 11a** and **c** indicates the number of cases predicted for the last 30 days.
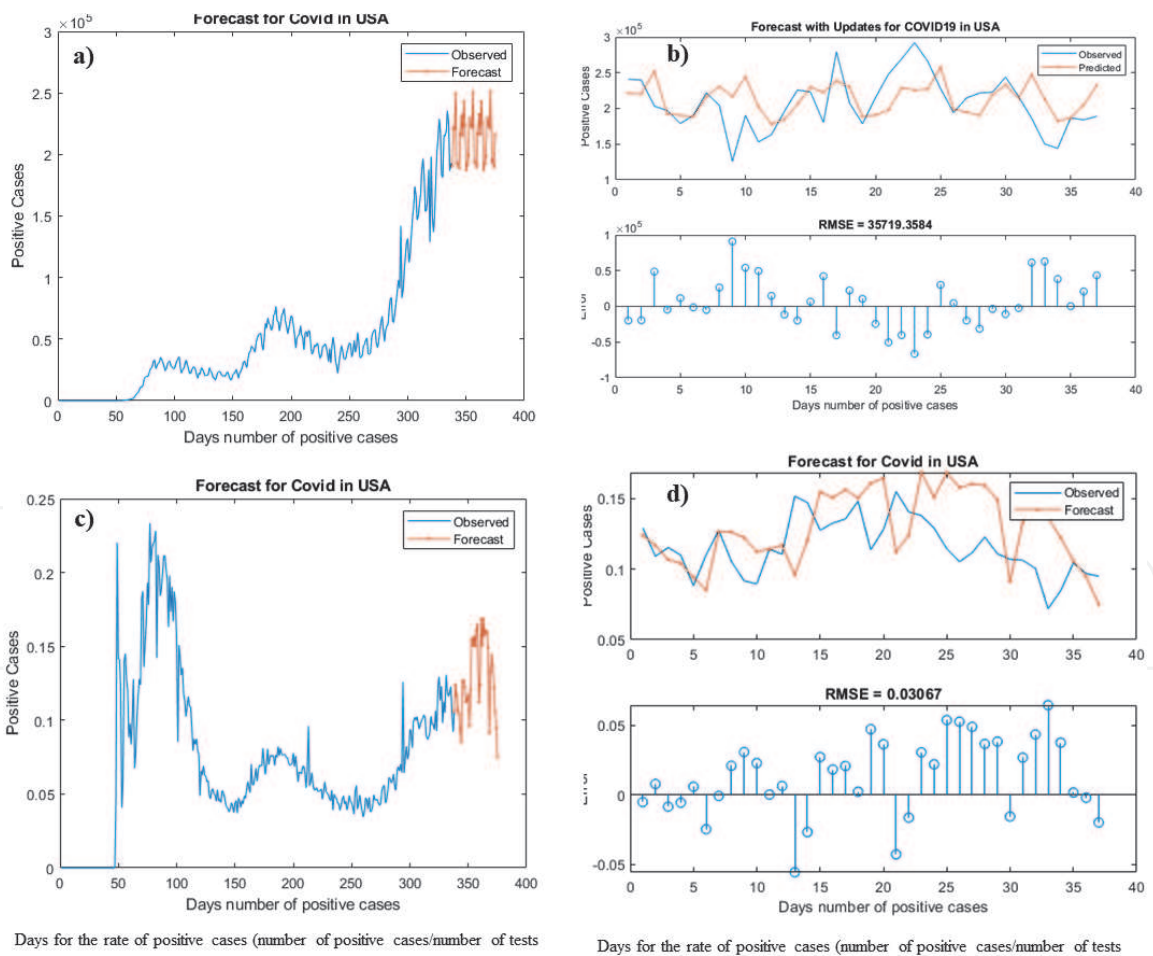

Figure 11.
Total daily number of positively tested COVID19 and the rate (positively tested/number of test) conducted in the USA. (a) Plot of the training time series of the number of positively tested COVID19 with the forecasted values, (b) compare the forecasted values of the number of positively tested with the test data set. This graph shows the total daily number of virus tests conducted in each state and of those tests, how many were positive each day. (c) Plot of the training time series of the rate of positively tested COVID19 (d) compare the forecasted values of the rate of positively tested with the rates in the test data set. The trend line in blue shows the actual number of positive cases and the trend line in red shows the number predicted for the last 38 days.

Example 1. MATLAB Codes and descriptions of the codes	
a. Data prepretion	**Descriptions**
numTimeStepsTrain = floor(0.95*numel(Tpositive)); dataTrain = Tpositive(1:numTimeStepsTrain+1); dataTest = Tpositive(numTimeStepsTrain+1:end);	Partition the training and test data. Train on the first 95% of the sequence and test on the last 5%
b. Define LSTM	**Descriptions**
numHiddenUnits = 300; layers = [...sequenceInputLayer(numFeatures) lstmLayer(numHiddenUnits) fullyConnectedLayer(numResponses) regressionLayer];	Define LSTM Network Architecture and create an LSTM regression network. Specify the LSTM layer to have 300 hidden units
c. Specify the training options	**Descriptions**
options = *trainingOptions('adam'*, ... 'MaxEpochs',1000, ... 'GradientThreshold',1, ... 'InitialLearnRate',0.005, ... 'LearnRateSchedule','piecewise', ... 'LearnRateDropPeriod',125, ... 'LearnRateDropFactor',0.2, ... 'Verbose',0, ... 'Plots','training-progress');	Training with Adam (adaptive moment estimation) To prevent the gradients from exploding, set the gradient threshold to 1. The software updates the learning rate every certain number of epochs by multiplying with a certain factor. Number of epochs for dropping the learning rate Factor for dropping the learning rate when 'LearnRateSchedule','piecewise'
d. Train the LSTM network	**Descriptions**
net = trainNetwork(XTrain,YTrain,layers,options);	Train the LSTM network with the specified training options
e. Initialize -training and loop over	
net = predictAndUpdateState(net,XTrain); [net,YPred] = predictAndUpdateState(net,YTrain (end)); numTimeStepsTest = numel(XTest); for i = 2:numTimeStepsTest [net,YPred(:,i)] = predictAndUpdateState(net,YPred (:,i-1),'ExecutionEnvironment','cpu'); end;	The training and update data Next, make the first prediction using the last time step of the training response YTrain(end). Loop over the remaining predictions and input the previous prediction to predictAndUpdateState.
f. Update Network State with observed values	**Descriptions**
net = resetState(net); net = predictAndUpdateState(net,XTrain);	Update Network State with Observed Values
g. Predict on each time step	
YPred = []; numTimeStepsTest = numel(XTest); for i = 1:numTimeStepsTest [net,YPred(:,i)] = predictAndUpdateState(net,XTest (:,i),'ExecutionEnvironment','cpu'); end	Predict on each time step. For each prediction, predict the next time step using the observed value of the previous time step.

*All descriptions are based on MATLAB 2020b and related examples from the MATLAB.

Table 1.
MATLAB codes and specification of cods for Example 1[*].

Example 2: Data from example 2 is from SEER 2017 for different age groups. SEER collects cancer incidence data from population-based cancer registries of the U.S. population. The SEER registries collect data on patient demographics, primary tumor site, tumor morphology, stage at diagnosis, and the first course of treatment, and they follow up with patients for vital status. The example given in this chapter

is the cancer type and non-cancer causes of death identified from the survey. https://seer.cancer.gov/data/. The steps for example 2 are summarized in **Table 2** (MATLAB 2020b) and because of space limitation, only the cloud of cancer from

Example 2. MATLAB Codes and descriptions of the codes	
a. Data prepretion	Descriptions
filename = "filename" data = readtable(filename,'TextType','string');	To import the text data as strings, specify the text type to be 'string'.
b. Partition data	Descriptions

crossval = cvpartition('DataName.
ClassVaribleName',0.30);
dataTrain = DataName(training(crossval),:);
dataValidation = DataName (test(crossval),:);
	Partition data into sets for training and validation. Partition the data into a training partition and a held-out partition for validation and testing. Specify the holdout percentage to be 30%.

c. Extract the text data	Descriptions
textDataTrain = dataTrain.TextVariableName_; textDataValidation = dataValidation. TextVariableName _; YTrain = dataTrain. TextVariableName; YValidation = dataValidation. TextVariableName;	Extract the text data and labels from the partitioned tables. Here TextVariableName is column name for the group variable (for example age group) and TextVariableName is the column name for the text variable (in this example the name of cancer types)
d. Create a cloud for the text	Descriptions
figure wordcloud(textDataTrain);	To check that you have imported the data correctly, visualize the training text data using a word cloud.
e. Preprocess Text Data	
documentsTrain = preprocessText (textDataTrain); documentsValidation = preprocessText (textDataValidation);	Preprocess the training data and the validation data using the preprocessText function.
f. Convert document to sequences	Descriptions

enc = wordEncoding(documentsTrain);
sequenceLength = 10;
XTrain = doc2sequence(enc,
documentsTrain,'Length',sequenceLength);
XTrain(1:5)
XValidation = doc2sequence(enc,
documentsValidation,'Length',sequenceLength);
	Encoding to convert the documents into sequences of numeric indices.

g. Create and Train LSTM	Descriptions

inputSize = 1;
embeddingDimension = 30;
numHiddenUnits = 200;
numWords = enc.NumWords;
numClasses = numel(categories(YTrain));
layers = [...
sequenceInputLayer(inputSize)
wordEmbeddingLayer(embeddingDimension, numWords)
lstmLayer(numHiddenUnits,'OutputMode','last')
fullyConnectedLayer(numClasses)
softmaxLayer
classificationLayer]
	Initialize the embedding weights

h. Specify training options	Descriptions
options = trainingOptions('adam', ... 'MiniBatchSize',30, ...	MiniBatchSize: Classifies data using mini batches of size

'GradientThreshold',8, … 'Shuffle','every-epoch', … 'ValidationData',{XValidation,YValidation}, … 'Plots','training-progress', … 'Verbose',false);	'GradientThreshold: Clip gradient values for the threshold

i. Train the LSTM network	**Descriptions**
net = trainNetwork(XTrain,YTrain,layers, options);	Train the LSTM network using the trainNetwork function.

j. Predict using new data	**Descriptions**
reportsNew = [… "The text definition here."];	Classify the event type of three new reports. Create a string array containing the new reports.

k. Preprocess Convert and Classify	**Descriptions**
documentsNew = preprocessText(reportsNew); XNew = doc2sequence(enc, documentsNew,'Length',sequenceLength); labelsNew = classify(net,XNew)	

All descriptions are based on MATLAB 2020b and related examples from the MATLAB.

Table 2.
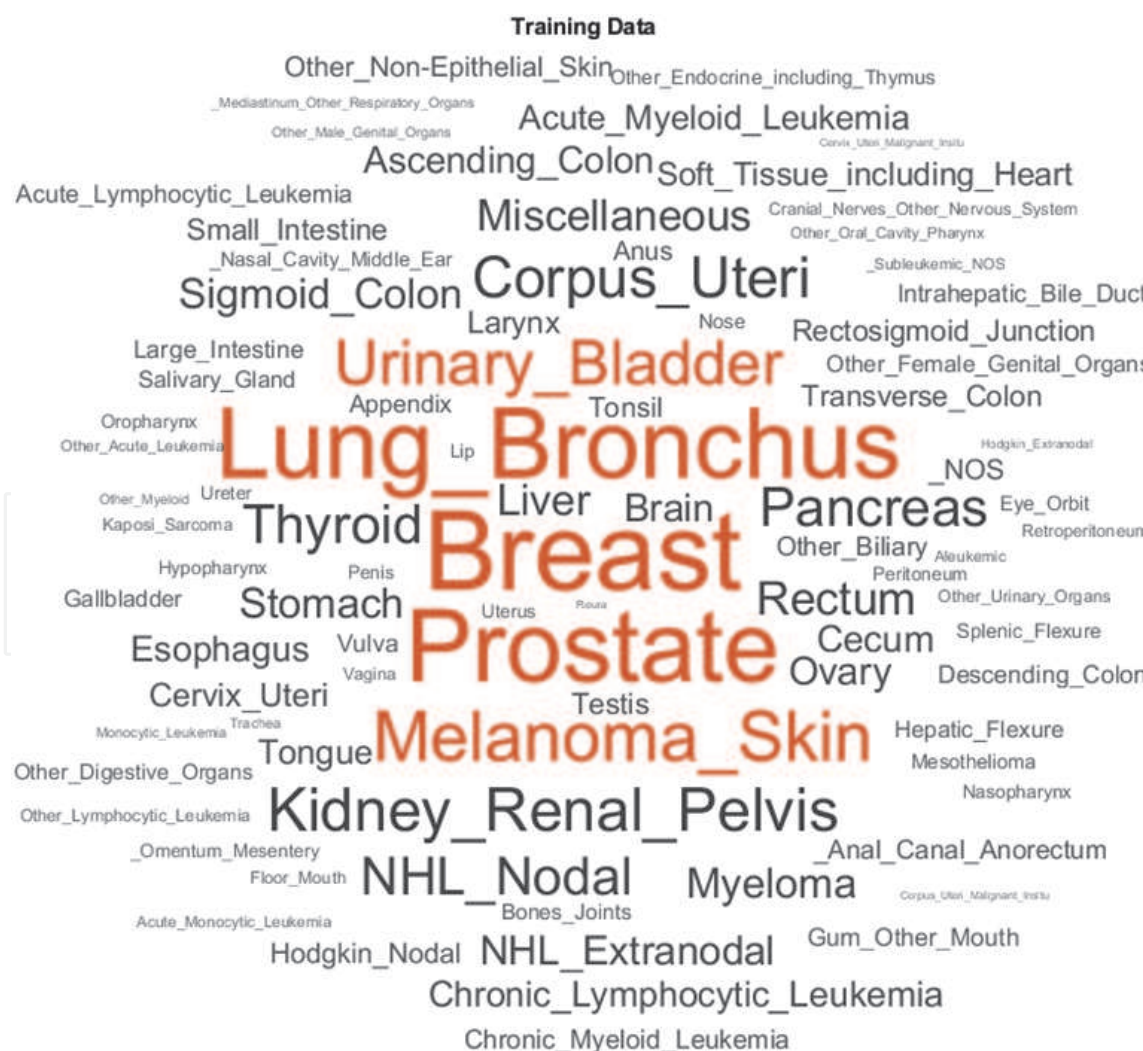*MATLAB codes and specification of cods for Example 2*.*

Figure 12.
*Visualizing the training data text file for SEER-2017 cancer types by age groups using a word cloud of LSTM. The MATLAB codes to create this figure are given in **Table 2**. The bigger the word, the more often diagnosed cancer type in 2017.*

age groups is illustrated in **Figure 12**. Here, in order to input the documents into an LSTM network, the "*wordEncoding(documentsTrain)*" is used. This code converts the name of diseases into sequences of numeric indices. The disease names (all types of text structure) in LSTM with MATLAB are performed in three consecutive steps: 1) tokenize the text 2) convert the text to lowercase and, 3) erase the punctuation. The function stops predicting when the network predicts the end-of-text character or when the generated text is 500 characters long.

5. Conclusions and future work

LSTM is a very powerful ANN architecture for disease subtypes, time series analyses, for the text generation, handwriting recognition, music generation, language translation, image captioning process. The LSTM approach is effective to make predictions as equal attention is provided for all input sequences by the information flows through the cell state. Because of the mechanism adopted, the small change in the input sequence does not harm the prediction accuracy done by LSTM. Future work on LSTM has several directions. Most LSTM architectures are designed to handle data evenly distributed between elapsed times (days, months, years, etc.) for the consecutive elements of a sequence. More studies are needed to improve the predictive ability of LSTM for nonconstant consecutive observations elapsed times. Moreover, further studies are needed for possible overfitting problems for training with smaller data sets. Rather than using early stopping to avoid the overfitting, Bayesian regularized approach would be more effective to ensure that the neural network halts training at the point where further training would result in overfitting. As the Bayesian regularized approach uses a different loss function with different hyperparameters, this approach demands costly computation resources.

Acknowledgements

Author details

Hayrettin Okut
The University of Kansas School of Medicine, Wichita, USA

*Address all correspondence to: hokut@kumc.edu

IntechOpen

References

[1] Okut, H., Wu, X-L., Rosa, JM. G., Bauck, S., Woodward, B., Schnabel, D. R., Taylor, F. J. and Gainola, D. Predicting expected progeny difference for marbling score in Angus cattle using artificial neural networks and Bayesian regression models. Genetics Selection Evolution 2013, 45:34 doi:10.1186/1297-9686-45-34.

[2] Okut H.,. Bayesian Regularized Neural Networks for Small n Big p Data, Artificial Neural Networks - Models and Applications, Joao Luis G. Rosa, IntechOpen, 2016. DOI: 10.5772/63256.

[3] Hochreiterand, S. and Schmidhuber, J., Long Short-Term Memory. Neural Computation. Volume 9 | Issue 8, 1997

[4] Schmidhuber, J. Deep Learning in Neural Networks: An Overview". Neural Networks.61: 85 17, 2015. arXiv: 1404.7828.

[5] Miotto, R., et al., "Deep patient: An unsupervised representation to predict the future of patients from the electronic health records," Sci. Rep., vol.6, no. 1, pp. 26094–26094, 2016.

[6] Choi, E., et al., "Doctor AI: Predicting clinical events via recurrent neural networks," in Proc. 1st Mach. Learn. Healthcare Conf., 2016, pp. 301–318.t

[7] Razavian, N., J. Marcus, and D. Sontag, "Multi-task prediction of disease onsets from longitudinal lab tests," in Proc. 1st Mach. Learn. Healthcare Conf., 2016, pp. 73–100.

[8] Yang Chao-Tung, Yuan-An, C.., Wei Chan, Y., Chia-Lin L., Yu-Tse T., Wei-Cheng C. and· Po-Yu, L. Liu (2020). Influenza-like illness prediction using a long short-term memory deep learning model with multiple open data sources. The Journal of Supercomputing (2020) 76:9303–9329 https://doi.org/10.1007/s11227-020-03182-5.

[9] S. Purushotham et al., "Benchmark of deep learning models on large healthcare mimic datasets," 2017.online available: h ttps://arxiv.org/abs/ 1710.08531

[10] Kim et al.,J. Y., "High risk prediction from electronic medical records via deep attention networks," Nov. 30, 2017. [Online]. Available: https: //arxiv.org/abs/1712.00010

[11] Ma, F., et al., "Dipole: Diagnosis prediction in healthcare via attention-based bidirectional recurrent neural networks," in Proc. 23rd ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining, Halifax, Canada, 2017, pp. 1903–1911.

[12] Nguyen, P., Tran, T. and Venkatesh, S. "Resset: A recurrent model for sequence of sets with applications to electronic medical records," in Proc. Int. Joint Conf. Neural Netw., Brazil, 2018, pp. 1–9.

[13] Maxwell, A., et al., "Deep learning architectures for multi-label classification of intelligent health risk prediction," BMC Bioinf., vol. 18, no. Suppl 14, pp. 523–523, 2017.

[14] Tingyan Wang, Yuanxin Tian , and Robin G. Qiu. Long Short-Term Memory Recurrent Neural Networks for Multiple Diseases Risk Prediction by Leveraging Longitudinal Medical Records. EEE Journal Of Biomedical And Health Informatics, Vol. 24, No. 8, August 2020 DO:1 0.1109/JBHI.2019.2962366.

[15] Baytas, I., Xiao, C., Zhang, X., Wang, F., Jain, K. A. and Zhou, Jiayu. Patient Subtyping via Time-Aware LSTM Networks. In Proceedings of KDD Halifax, NS, Canada, 2017..DOI: 10.1145/3097983.3097997.

[16] Okut, H., Gianola, D., Rosa, J. G., Weigel, K. Prediction of body mass

index in mice using dense molecular markers and a regularized neural network. Genetics Research (Cambridge). 2011. 93:189–201.

[17] Lipton, C. Z., Berkowitz, J. and Elkan, C. A Critical Review of Recurrent Neural Networks for Sequence Learning. arXiv:1506.00019v4.

[18] Colah, C. Understating LSTM Network. https://colah.github.io/posts/ 2015-08-Understanding-LSTMs/

[19] Ali. M. A., Zhuang, H., Ibrahim, A., Rehman, O., Huang, M and Wu, A. A Machine Learning Approach for the Classification of Kidney Cancer Subtypes Using miRNA. Genome Data. Appl. Sci. 2018, 8, 2422; doi:10.3390/ app8122422.

[20] https://www.geeksforgeeks. org/lstm-derivation-of-back-propaga tion-through-time/?ref=lbp. 2020.

[21] Gers, F. A., Schmidhuber, J. and Cummins, F. Learning to forget: Continual prediction with LSTM. In Proc. ICANN'99, Int. Conf. on Artificial Neural Networks, Vol. 2, pp. 850–855, 2000. Edinburgh, Scotland. IEE, London. Extended version submitted to Neural Computation.

[22] Kyunghyun, C., van Merrienboer, Gulcehre, Caglar, F., Dzmitry, B., Fethi B.,Holger, H. and Yoshua, B. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation, 2014.arXiv: 1406.1078.

[23] Krause, B., Murray, I. and Renals S. Multiplicative LSTM for sequence modelling., 2017. arXiv:1609.07959v3

[24] Wu, Y., Schuster,M., Chen, Z., Le V. Q., Norouzi, M., Macherey, W., Krikun, M, Cao, Y., Gao, Q., Macherey, K., Klingner, J., Shah, A., Johnson, M., Liu, X., Kaiser, L., Gouws, S., Kato, Y., Taku, K., Kazawa, H., Stevens, K.,

Kurian, G., Patil, N., Wang, W., Young, C., Smith, J., Riesa, J., Rudnick, A., Vinyals, O., Corrado, G., Hughes, M. and Dean, J. Google's Neural Machine Translation System: Bridging tshe Gap between Human and Machine. Translation.2017, arXiv:1609.08144v2