Aston University
BIRMINGHAM UK

CS4755_P2_Mathematics for AI

Mathematical Foundations of LSTM for Sequential Data

Student Name : Kartik Padalkar
Student ID : 230388479

Table of Contents

1.Introduction:

1.1.Applications:

Sequential data learning tasks such as prediction, classification, sequence generation, etc. have become crucial. Sequential data, at its core is referred as the data which is recorded in a specific order where the order of data points is important for interpretation or analysis[1]. Sequential data exists in various domains such as NLP, time series, audio processing, etc[2]. Various methodologies are prominent for sequential leaning tasks, such as Recurrent neural networks(RNN) which was first conceptualized in 1986 [3][4]. In recent years many advancements have made it possible to use RNN in practical applications[5]. Long Short Term memory (LSTM) networks are a type of Recurrent Neural Network (RNN) which are widely used for sequential data analysing and predicting tasks because of their ability to process long sequences, capable of learning and remembering patterns which is important for tasks that involve temporal dependencies[6][7].

1.2.Problem :

Recurrent Neural Networks are a type of neural networks which are used in learning various sequential tasks. Although they struggle when long sequences are introduced. RNN are prone to the problem of exploding and vanishing gradients which has long sequences of data[8]. Gradients may vanish or explode during backpropagation due to repeated multiplicative effect on derivatives. For example, activations like sigmoid, the derivatives are small, bounded between 0 and 0.25 globally. Multiplying many small values repeatedly results in vanishing gradients, as the gradients shrink rapidly backwards. Alternatively, with activations having derivatives > 1 like, gradients grow unstably as signals get multiplied across layers. This exponential growth causes exploding gradients, making optimization difficult.The core issue is having derivatives that are consistently < 1 or > 1 across layers.[9][10].

In this report, comprehensive mathematical foundations of LSTM, explaining key features for applications in sequential data modelling and prediction tasks and how they address the problem of RNN with help of numerical example are presented and strengths as well as weaknesses are discussed.

2.Long short-term memory:

LSTM is a type of RNN which were designed to deal with this problem. LSTMs address this issue by introducing gating mechanisms which has more complex memory handling which manages the flow of information. In a LSTM architecture two main units are memory cell that can retain its state over time and non-linear gating unit which control flow of information in and out of the cell[11]. Numerous advancements have been made to the architecture since its introduction and are incorporated in multiple learning tasks.

2.1. LSTM architecture

Unlike RNN which consists a simple unit to unroll, LSTM has a much more complicated architecture. LSTM cell uses sigmoid activation function, which basically converts any input between 0 and 1 and tanh activation function which converts any input between -1 and 1.

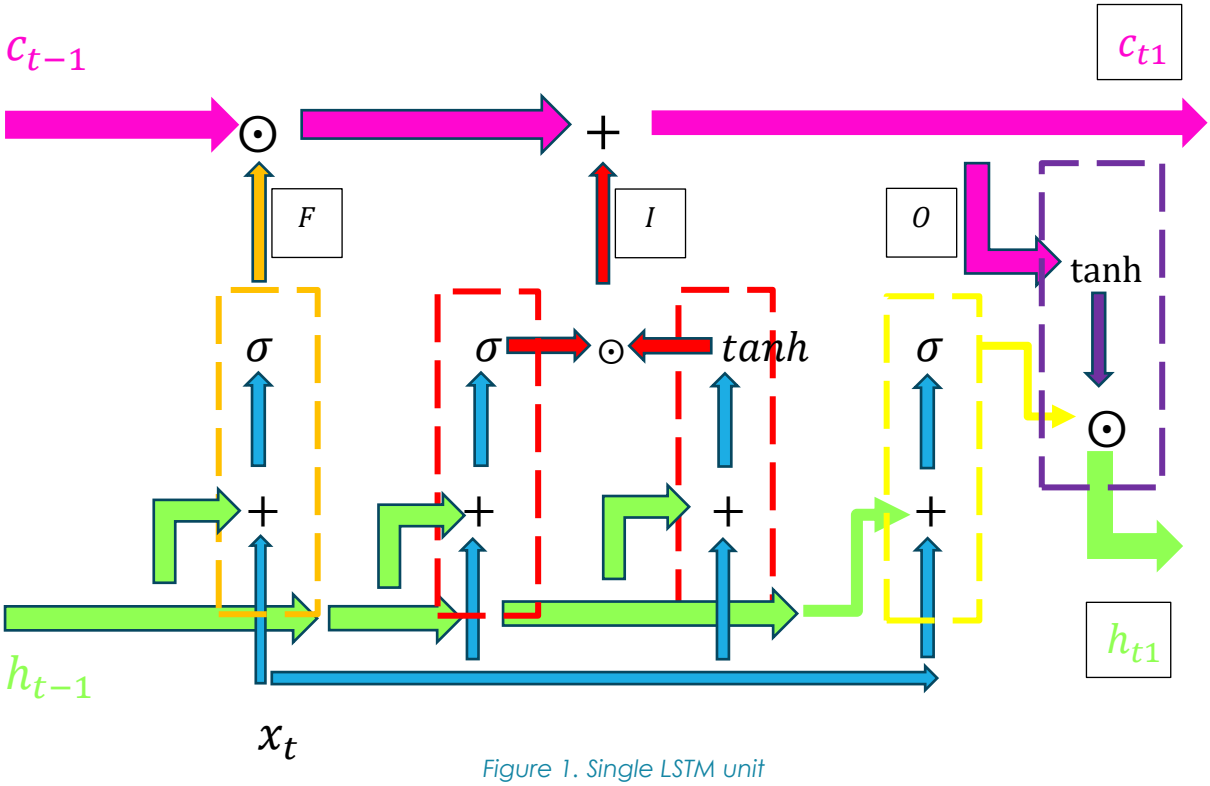

Figure 1. Single LSTM unit

Notations:

c_t = Cell state
h_t = Hidden state
F = Forget gate
I = Input gate
O = Output gate
x_t = Input
$\odot$ = Element wise multiplication
σ = Sigmoid activation function
tanh = tan h activation function

2.2.Mathematical foundations.

As illustrated in figure 1. LSTM has three main gates - forget gate, input gate and output gate.

2.2.1. Forget Gate

The forget gate uses a mathematical operation which acts like a regulator for the information given by the weighted sum of input and hidden state and a bias term. By employing a sigmoid function, the resulting value ranges between 0 and 1. For example, sigmoid activation function is also given by $f(x) = \frac{e^x}{e^{x+1}}$, let's consider value given by weighted sum of input and output is $x = 10 \therefore e^{10} = 2206.465 \therefore f(x) = 0.999$, which is close to 1 this indicates that the cell state can keep its memory because output of sigmoid function is multiplied by previous cell state. Similarly if $x = -10$ the resulting value given by the sigmoid function is 0.00004 which is close to 0 this means forgetting the past memory of the cell state. Because these values range between 0 and 1, it indicates the degree of forgetting or remembering past values respectively.

Output of the forget gate is computed by –

$$F = \sigma[(W_{xF} \cdot x_t) + (W_{hF} \cdot h_{t-1}) + b_F] \quad (1)$$

Where, W is weights and b is the bias for the respective gate and t is the time step.

2.2.2. Input Gate

The input gates main purpose is to decide which new information should be added to the cell state. It uses sigmoid activation function same as before and a new tanh function is introduced in the modulation gates.. The tanh function is given by $f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$, which converts any value to $-1\ or\ 1$,for example if $x = 10$ the value will be 0.999 and if $x = -10$ then its value will be -0.999. Hence, this gives ability to modulation gate with $\tanh function$ to add a potential memory to the cell state. Modulation gate with sigmoid function has the same task as before.

Output of the Input gate is computed by –
In input gate there are two modulation gates are denoted by g & a respectively as shown in figure 1

$$g = \sigma\big[(W_{xg} \cdot x_t) + (W_{hg} \cdot h_{t-1}) + b_g\big] \quad (2)$$

$$a = \tanh[(W_{xa} \cdot x_t) + (W_{ha} \cdot h_{t-1}) + b_a] \quad (3)$$

$$\because I = a \odot g \quad (4)$$

2.2.3. Cell State

The cell state functions as a channel to preserve and transfer information across time steps. Since there are no weights and biases that can directly modify the information it allows efficient learning and retention of long term dependencies of sequential data without causing the gradients to vanish or explode. However, it is updated through addition and multiplication, influenced by forget and input gates.

So the updated cell state is computed by –

$$c_{t1} = (c_{t-1} \odot F) + I \quad (5)$$

2.2.4. Output Gate

The output value of Output gate regulates the information flow of cell state to calculate the final hidden state output by incorporating the sigmoid function. This ensures that relevant data from the cell state is appropriately influenced to the final hidden state output in sequential modelling and prediction.

Output of the output gate is derived by –

$$O = \sigma[(W_{xO} \cdot x_t) + (W_{hO} \cdot h_{t-1}) + b_O] \quad (6)$$

2.2.5. Hidden State

The tanh function in this is applied to the cell state which keeps the values appropriately scaled between -1 and 1 ensuring the stability of the network, enabling LSTM to learn effectively from sequential data without causing numerical instability.

The whole updated hidden state is calculated by –

$$h_{t1} = \tanh(c_{t1}) \odot O \tag{7}$$

2.2.6. Equations for Backward propagation:

For Backward propagation a loss function is defined, in this case MSE is used.

It is defined as : $E(x, \hat{x}) = \frac{(x-\hat{x})^2}{2}$ (8)

By derivation with respect to x : $\frac{dE}{dx} = \partial x E = x - \hat{x}$ (9)

Here, $x = true\ value\ \&\ \hat{x} = predicted\ value$, in this case it is the output of updated hidden state value.

So, to perform a backward pass, gradients with respect to forget gate, output gate, input gate and hidden state with their respective weights are needed, gradients are derived using chain rule.

Gradient with respect to $h_t =$ $\delta h_t = \Delta t \quad + \Delta h_{t+1}$ (10)

Gradients with respect to -

$Cell\ state\ c_t =$ $\delta c_t = \delta h_t \odot\ O \odot (1 - \tanh^2(c_t))$ (11)

$Input\ gate\ unit\ a_t =$ $\delta a_t = \delta c_t \odot g_t \odot (1 - a_t^2)$ (12)

$Input\ gate\ unit\ g_t =$ $\delta g_t = \delta c_1 \odot a_1 \odot g_1 \odot (1 - g_1)$ (13)

$Forget\ gate\ F_t =$ $\delta F_t = \delta c_t \odot c_{t-1} \odot F_t \odot (1 - F_t)$ (14)

$Output\ gate\ O_t =$ $\delta O_t = \delta h_t \odot \tanh(c_t) \odot\ O_t \odot (1 - O_t)$ (15)

$Input\ x_t =$ $\delta x_t = W_x^T\ .\ \delta gates_t$ (16)

$Gradient\ update\ of\ hidden\ state\ for\ backward\ pass -$

$$\Delta h_{t-1} = W_h^T \cdot \delta gates_t \tag{17}$$

The final gradient update to the respective weights of input and hidden state and biases can be calculated by:

$$\delta W_x = \sum_{t=0}^{T} \delta gates_t \otimes x_t \tag{18}$$

$$\delta W_h = \sum_{t=0}^{T-1} \delta gates_{t+1} \otimes h_t \tag{19}$$

$$\delta b = \sum_{t=0}^{T} \delta gates_{t+1} \tag{20}$$

3. Numerical Example :

Inputs for the deep learning models are generally given by a input matrix with a corresponding label. For example let's consider univariate sequential data denoted as dt with values = [a, b, c, d] then input 1 that is $dt_1 = [a, b]$ with corresponding label $[c]$ and similarly input 2 that is $dt_2 = [b, c]$ with label $[d]$. During backpropagation, loss is calculated between the predicted output and the label which serves as the target variable[12][13]. Then it updates weights and biases until the model converges and learns the

temporal dependencies and patterns within the data. The inputs are not restricted as the given example it can be multivariate or covariates which demonstrates the flexibility of the model.

In this example, the main motive is not to forecast but to show how the inputs and gradients flow with help of numbers.

The sequential data in this example is considered as :

Data = [2, 4, 6, 8, 10] …(Univariate sequential data)

x0 = $\begin{bmatrix}2\\4\end{bmatrix}$; y0 = [6] …(time step 0)

x1 = $\begin{bmatrix}6\\8\end{bmatrix}$; y1 = [10] ...(time step 1)

Random weights for the respective input(x) and hidden state(h) and biases(b)are considered for this example.

$W_{xa} = [0.34 \quad 0.6]$ $W_{ha} = [0.75]$ $b_a = [0.61]$

$W_{xg} = [0.47 \quad 0.52]$ $W_{hg} = [0.69]$ $b_g = [0.29]$

$W_{xF} = [0.2 \quad 0.59]$ $W_{hF} = [0.31]$ $b_F = [0.18]$

$W_{xO} = [0.64 \quad 0.93]$ $W_{hO} = [0.57]$ $b_O = [0.31]$

$$W_x = \begin{bmatrix}0.34 & 0.6\\0.47 & 0.52\\0.2 & 0.59\\0.64 & 0.93\end{bmatrix} \quad W_h = \begin{bmatrix}0.75\\0.69\\0.31\\0.57\end{bmatrix} \quad b = \begin{bmatrix}0.61\\0.29\\0.18\\0.31\end{bmatrix}$$

3.1. Forward Propagation:

At time step 0 , cell state and hidden state does not hold any values, therefore,

$c_{t-1} = [0]$

$h_{t-1} = [0]$

Forward Pass (time step 0)

From equation (1) …

$$F_0 = \sigma[(W_{xF} \cdot x_0) + (W_{hF} \cdot h_{t-1}) + b_F]$$

$$= \sigma\left([0.2 \quad 0.59]\begin{bmatrix}2\\4\end{bmatrix}\right) + ([0.31][0]) + [0.18]$$

$$\therefore F_0 = 0.94978873 \qquad (21)$$

From equation(2)…

$$g = \sigma\left[(W_{xg} \cdot x_t) + (W_{hg} \cdot h_{t-1}) + b_g\right]$$

$$= \sigma\left([0.47\ 0.52]\begin{bmatrix}2\\4\end{bmatrix}\right) + ([0.69][0]) + [0.29]$$

$$\therefore g_0 = 0.96477028 \qquad (22)$$

From equation(3)…

$$a_0 = \tanh[(W_{xa} \cdot x_t) + (W_{ha} \cdot h_{t-1}) + b_a]$$

$$= \tanh\left([0.34\ 0.6]\begin{bmatrix}2\\4\end{bmatrix}\right) + [0.75][0] + [0.61]$$

$$\therefore a_0 = 0.99875358 \tag{23}$$

From equation(6)…

$$O_0 = \sigma[(W_{xO} \cdot x_t) + (W_{hO} \cdot h_{t-1}) + b_O]$$

$$= \sigma(\left([0.64\ 0.93]\begin{bmatrix}2\\4\end{bmatrix}\right) + ([0.57][0]) + [0.31]$$

$$\therefore O_0 = 0.99508238 \tag{24}$$

From equations(5)(22)(23)…

$$c_0 = (c_{t-1} \odot F_0) + I$$

$$= (0)(0.82635335) + (0.96477028)(0.998753586)$$

$$\therefore c_0 = 0.96356777 \tag{25}$$

From equations(7)and (24)…

$$h_0 = \tanh(c_0) \odot O_0$$

$$= \tanh(0.835173)\ (\ 0.94321382)$$

$$\therefore h_0 = 0.74219618 \tag{26}$$

Similarly, Using NumPy Forward pass to time step 1 is calculated, the only difference is that, from equation (26) we have updated hidden state value to pass forward to time step 1…

[2]:
```
F_1 = sigmoid((W_xF @ x1) + (W_hF @ h_0) + bF)
a_1 = np.tanh((W_xa @ x1) + (W_ha @ h_0) + ba)
g_1 = sigmoid((W_xg @ x1) + (W_hg @ h_0) + bg)
O_1 = sigmoid((W_xO @ x1) + (W_hO @ h_0) + bO)
c_1 = (F_1 @ c_0) + (a_1 @ g_1)
h_1 = (O_1 @ np.tanh(c_1))
```

Calculated values at Forward pass (time step 1) are as follows :

$$F_1 = 0.99822128 \tag{27}$$

$$a_1 = 0.99999978 \tag{28}$$

$$g_1 = 0.99958305 \tag{29}$$

$$O_1 = 0.99999394 \tag{30}$$

$$c_1 = 1.96143667 \tag{31}$$

$$h_1 = 0.96119348 \tag{32}$$

3.2. Backward Propagation

Backward propagation (time step 1)

Since there is no future time step, no gradients are passed backwards

$\delta c_2 = [0]$

$F_2 = [0]$

$\Delta h_{t+1} = [0]$

To proceed with backpropagation loss is calculated using equation(9) with respect to time step 1

$$\Delta t_1 = \partial_x E = h_1 - y1$$

$$= 0.96119348 - 10$$

$$\Delta t_1 = -9.03880652 \quad (33)$$

From equation (10)

$$\delta h_1 = \Delta t_1 + \Delta h_{t+1}$$

$$= (-9.03880652) + (0)$$

$$\therefore \delta h_1 = -9.03880652 \quad (34)$$

From equation (11)(30)(34)

$$\delta c_1 = \delta h_1 \odot O_1 \odot (1 - \tanh^2(c_1)) + \delta c_{t2} \odot F_2$$

$$= (-9.03880652)(0.99999394)(0.07609589) + (0)(0)$$

$$\therefore \delta c_1 = -0.68781188 \quad (35)$$

From equation (12)(13)(28)(35)

$$\delta a_1 = \delta c_1 \odot g_1 \odot (1 - a_1^2)$$

$$= (-0.68781188)(0.99958305)(4.44195913e-07)$$

$$\therefore \delta a_1 = -3.05395836e-07 \quad (36)$$

From equation (13)(28)(29)(35)

$$\delta g_1 = \delta c_1 \odot a_1 \odot g_1 \odot (1 - g_1)$$

$$= (-0.68781188)(0.99999978)(0.99958305)(0.00041695)$$

$$\therefore \delta g_1 = -2.86666918e-04 \quad (37)$$

From equation (14)(25)(27)

$$\delta F_1 = \delta c_1 \odot c_0 \odot F_1 \odot (1 - F_1)$$

$$= (-0.68781188)(0.96356777)(0.99822128)(0.00177872)$$

$$\therefore \delta F_1 = -1.17675598e-03 \quad (38)$$

From equation (15)(30)(31)(34)

$$\delta O_t = \delta h_1 \odot \tanh(c_1) \odot O_1 \odot (1 - O_1)$$

$$= (-9.03880652)(0.96119931)(0.99999394)(6.06450348e-06)$$

$$\therefore \delta O_1 = -5.26886602e-05 \quad (39)$$

Created an array of all the gradients of the gates outputs for further calculation.

$$\delta outputs_1 = \begin{bmatrix} -3.05395836e-07 \\ -2.86666918e-04 \\ -1.17675598e-03 \\ -5.26886602e-05 \end{bmatrix} \quad (40)$$

The gradient of loss with respect to x_t is calculated by transposing the input weight matrix W_x and multiplying with gradients of the outputs at the particular time step $\delta output$

$$\delta x_1 = \begin{bmatrix} 0.34 & 0.47 & 0.2 & 0.64 \\ 0.6 & 0.52 & 0.59 & 0.93 \end{bmatrix} \begin{bmatrix} -3.05395836e-07 \\ -2.86666918e-04 \\ -1.17675598e-03 \\ -5.26886602e-05 \end{bmatrix} = \begin{bmatrix} -0.00040391 \\ -0.00089254 \end{bmatrix} \quad (41)$$

Similarly to calculate gradient of loss with respect to h_t to pass backward $(t-1)$ uses the same formula with transposed hidden state weight matrix W_h.

$$\Delta h_0 = [0.75 \quad 0.69 \quad 0.31 \quad 0.57] \begin{bmatrix} -3.05395836e-07 \\ -2.86666918e-04 \\ -1.17675598e-03 \\ -5.26886602e-05 \end{bmatrix} = -0.00059286 \quad (42)$$

Likewise Backward propagation at time step 0 can be calculated. Implemented NumPy to calculate the values.

[3]:
```
delta_t0 = h_0 - y0

d_h0 = delta_t0 + delta_h_0
d_c0 = d_h0.dot(O_0).dot(1 - np.square(np.tanh(c_0))) + d_c1.dot(F_1)
d_a0 = d_c0.dot(g_0).dot(1 - np.square(a_0))
d_g0 = d_c0.dot(a_0).dot(g_0).dot(1 - g_0)
d_F0 = d_c0.dot(ct_minus_1).dot(F_0).dot(1 - F_0)
d_O0 = d_h0.dot(np.tanh(c_0)).dot(O_0).dot(1 - O_0)
```

Calculated values by NumPy at Backward pass at time step 0 are as follows:

$$\Delta t_0 = -5.25780382 \quad (43)$$

$$\delta h_0 = -5.25839668 \quad (44)$$

$$\delta c_0 = -3.00819649 \quad (45)$$

$$\delta a_0 = -0.00723028 \quad (46)$$

$$\delta g_0 = -0.10211691 \tag{47}$$

$$\delta F_0 = [-0.] \tag{48}$$

$$\delta O_0 = -0.01919232 \tag{49}$$

$$\delta outputs_0 = \begin{bmatrix} -0.00723028 \\ -0.10211691 \\ 0 \\ -0.01919232 \end{bmatrix} \tag{50}$$

Gradient of loss with respect to x_t :

$$\delta x_0 = \begin{bmatrix} 0.34 & 0.47 & 0.2 & 0.64 \\ 0.6 & 0.52 & 0.59 & 0.93 \end{bmatrix} \begin{bmatrix} -0.00723028 \\ -0.10211691 \\ 0 \\ -0.01919232 \end{bmatrix} = \begin{bmatrix} -0.06273632 \\ -0.07528782 \end{bmatrix} \tag{51}$$

Gradient of loss with respect to h_t, $for\ h_{t-1}$:

$$\Delta h_{t-1} = [0.75 \quad 0.69 \quad 0.31 \quad 0.57] \begin{bmatrix} -0.00723028 \\ -0.10211691 \\ 0 \\ -0.01919232 \end{bmatrix} = -0.086823 \tag{52}$$

Gradient updates with respect to weights and biases:

[4]:
```
d_Wx = np.add(d_outputs_0.dot(x0.reshape(1, 2)), d_outputs_1.dot(x1.reshape(1,
↪2)))
d_Wh = d_outputs_1 @ (h_0)
d_b = np.add(d_outputs_0, d_outputs_1)
```

$$\delta W_x = \begin{bmatrix} -0.0144624 & -0.02892358 \\ -0.20595381 & -0.41076096 \\ -0.00706054 & -0.00941405 \\ -0.03870076 & -0.07719077 \end{bmatrix} \tag{53}$$

$$\delta W_h = \begin{bmatrix} -2.26663622e-07 \\ -2.12763091e-04 \\ -8.73383791e-04 \\ -3.91053221e-05 \end{bmatrix} \tag{54}$$

$$\delta b = \begin{bmatrix} -0.00723059 \\ -0.10240357 \\ -0.00117676 \\ -0.019245 \end{bmatrix} \tag{55}$$

Updated weights and biases using Stochastic Gradient Descent with learning rate 0.1 :

[5]:
```
Wx_new = Wx - 0.1 * d_Wx
Wh_new = Wh - 0.1 * d_Wh
b_new = b - 0.1 * d_b
```

$$W_{x_{new}} = \begin{bmatrix} 0.34144624 & 0.60289236 \\ 0.49059538 & 0.5610761 \\ 0.20070605 & 0.5909414 \\ 0.64387008 & 0.93771908 \end{bmatrix} \quad (56)$$

$$W_{h_{new}} = \begin{bmatrix} 0.75000002 \\ 0.69002128 \\ 0.31008734 \\ 0.57000391 \end{bmatrix} \quad (57)$$

$$b_{new} = \begin{bmatrix} 0.61072306 \\ 0.30024036 \\ 0.18011768 \\ 0.3119245 \end{bmatrix} \quad (58)$$

This completes one iteration by updating the weights and biases of an LSTM cell.

In summary, with the help of gated units the forward and backward pass equations implement calculations in the input, output and forget gates using element-wise multiplication. These gates help modulate the flow of information and gradients which basic RNN lacks. Cell state equations demonstrate the use of cell state that transfers information across timesteps, this recurrent connection allows direct gradient flow through the cell state, whereas simple RNNs have only a single hidden state. LSTM uses both sigmoid and tanh activation functions.

The related topics covered in the lectures are partial derivatives, chain rule, matrix multiplication and transpose of a matrix.
For full coding implementation see appendix.

4. Discussion:

LSTMs have demonstrated state-of-the-art performance across various sequence modelling and prediction tasks, due to its ability to learn long term dependencies. Due to this it can learn complex temporal relationships in series of data. Moreover, in natural language processing it has improved performance on task like language translation, text generation, sentiment analysis compared to traditional n-gram and bag of words models[14]. Furthermore, in speech recognition it can learn characteristic audio patterns associated with phonemes and words improving transcription accuracy by leveraging broader acoustic context compared to HMMs[11]. Despite the strengths it has distinct weaknesses, high computational complexity due to multiple matrix operations for gates, cell updates. It is still susceptible to exploding gradients as suggested in paper [15] section "3.4 Training details."
In conclusion, while LSTM networks present exceptional performance in capturing long-term dependencies across various sequential tasks, their computational complexity and vulnerability to exploding gradients remain challenges. Emerging models like Transformer architectures, such as the BERT model in natural language processing and TCN (Temporal Convolutional Networks) in sequence modelling, offer promising alternatives by addressing some limitations of LSTMs and advancing the field of sequential modelling.

5. References:

[1]B. Shickel and P. Rashidi, "Sequential Interpretability: Methods, Applications, and Future Direction for Understanding Deep Learning Models in the Context of Sequential Data." Available: https://arxiv.org/pdf/2004.12524.pdf

[2]B. L. Nazarathy Sarat Moka, and Yoni, *8 Sequence Models | The Mathematical Engineering of Deep Learning (2021)*. Available: https://deeplearningmath.org/sequence-models.html#recurrent-neural-network

[3]D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, Oct. 1986, doi: https://doi.org/10.1038/323533a0.

[4]Z. Lipton, J. Berkowitz, and C. Elkan, "A Critical Review of Recurrent Neural Networks for Sequence Learning," 2015. Available: https://arxiv.org/pdf/1506.00019.pdf

[5]L. Deng, "Deep Learning: Methods and Applications," Foundations and Trends® in Signal Processing, vol. 7, no. 3–4, pp. 197–387, 2014, doi: https://doi.org/10.1561/2000000039.

[6]S. Hochreiter and J. Schmidhuber, "Long Short-Term Memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997, doi: https://doi.org/10.1162/neco.1997.9.8.1735.

[7]K. Greff, R. K. Srivastava, J. Koutnik, B. R. Steunebrink, and J. Schmidhuber, "LSTM: A Search Space Odyssey," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 28, no. 10, pp. 2222–2232, Oct. 2017, doi: https://doi.org/10.1109/tnnls.2016.2582924.

[8]Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE Transactions on Neural Networks*, vol. 5, no. 2, pp. 157–166, Mar. 1994, doi: https://doi.org/10.1109/72.279181.

[9]S. Hochreiter, "(PDF) The Vanishing Gradient Problem During Learning Recurrent Neural Nets and Problem Solutions," *ResearchGate*. https://www.researchgate.net/publication/220355039_The_Vanishing_Gradient_Problem_During_Learning_Recurrent_Neural_Nets_and_Problem_Solutions

[10]R. Pascanu, T. Mikolov, and Y. Bengio, "On the difficulty of training Recurrent Neural Networks." Available: https://arxiv.org/pdf/1211.5063.pdf

[11]A. Graves, A. Mohamed, and G. Hinton, "Speech recognition with deep recurrent neural networks," *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, 2013, doi: https://doi.org/10.1109/icassp.2013.6638947.

[12]H. Hewamalage, C. Bergmeir, and K. Bandara, "Recurrent Neural Networks for Time Series Forecasting: Current status and future directions," *International Journal of Forecasting*, vol. 37, no. 1, pp. 388–427, Jan. 2021, doi: https://doi.org/10.1016/j.ijforecast.2020.06.008.

[13]J. Brownlee, "How to Develop LSTM Models for Time Series Forecasting," *Machine Learning Mastery*, Nov. 13, 2018. https://machinelearningmastery.com/how-to-develop-lstm-models-for-time-series-forecasting/

[14]G. Van Houdt, C. Mosquera, and G. Nápoles, "A review on the long short-term memory model," *Artificial Intelligence Review*, vol. 53, no. 8, May 2020, doi: https://doi.org/10.1007/s10462-020-09838-1.

[15]I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to Sequence Learning with Neural Networks," *arXiv.org*, 2014. https://arxiv.org/abs/1409.3215

6. Appendix:

```
import numpy as np

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

W_xa = np.array([0.34, 0.6]).reshape(1, 2)
W_xg = np.array([0.47, 0.52]).reshape(1, 2)
W_xF = np.array([0.2, 0.59]).reshape(1, 2)
W_xO = np.array([0.64, 0.93]).reshape(1, 2)

W_ha = np.array(0.75).reshape(1, 1)
W_hg = np.array(0.69).reshape(1, 1)
W_hF = np.array(0.31).reshape(1, 1)
W_hO = np.array(0.57).reshape(1, 1)

ba = np.array(0.61).reshape(1, 1)
bg = np.array(0.29).reshape(1, 1)
bF = np.array(0.18).reshape(1, 1)
bO = np.array(0.31).reshape(1, 1)

Wx = np.vstack((W_xa, W_xg, W_xF, W_xO))
Wh = np.vstack((W_ha, W_hg, W_hF, W_hO))
b = np.vstack((ba, bg, bF, bO))

ct_minus_1 = np.array(0).reshape(1, 1)
ht_minus_1 = np.array(0).reshape(1, 1)
x0 = np.array([2, 4]).reshape(2, 1)
x1 = np.array([6, 8]).reshape(2, 1)
y0 = np.array(6).reshape(1, 1)
y1 = np.array(10).reshape(1, 1)

# Forward propagation time step = 0
a_0 = np.tanh((W_xa @ x0) + (W_ha @ ht_minus_1) + ba)
g_0 = sigmoid((W_xg @ x0) + (W_hg @ ht_minus_1) + bg)
F_0 = sigmoid((W_xF @ x0) + (W_hF @ ht_minus_1) + bF)
O_0 = sigmoid((W_xO @ x0) + (W_hO @ ht_minus_1) + bO)
c_0 = (F_0 @ ct_minus_1) +  (a_0 @ g_0)
h_0 = (O_0 @ np.tanh(c_0))

# Forward propagation time step = 1
a_1 = np.tanh((W_xa @ x1) + (W_ha @ h_0) + ba)
g_1 = sigmoid((W_xg @ x1) + (W_hg @ h_0) + bg)
F_1 = sigmoid((W_xF @ x1) + (W_hF @ h_0) + bF)
O_1 = sigmoid((W_xO @ x1) + (W_hO @ h_0) + bO)
c_1 = (F_1 @ c_0) + (a_1 @ g_1)
h_1 = (O_1 @ np.tanh(c_1))
```

```
# Backward propagation time step = 1
# future value
d_c2 = np.array(0).reshape(1, 1)
F_2 = np.array(0).reshape(1, 1)

delta_t1 = h_1 - y1
delta_h_t1 = 0

d_h1 = delta_t1 + delta_h_t1
d_c1 = d_h1.dot(O_1).dot(1 - np.square(np.tanh(c_1))) + d_c2.dot(F_2)
d_a1 = d_c1.dot(g_1).dot(1 - np.square(a_1))
d_g1 = d_c1.dot(a_1).dot(g_1).dot(1 - g_1)
d_F1 = d_c1.dot(c_0).dot(F_1).dot(1 - F_1)
d_O1 = d_h1.dot(np.tanh(c_1)).dot(O_1).dot(1 - O_1)

d_outputs_1 = np.vstack((d_a1, d_g1, d_F1, d_O1))

d_x1 = Wx.T @ d_outputs_1
delta_ht_0 = Wh.T @ d_outputs_1

# Backward propagation time step = 0
delta_t0 = h_0 - y0

delta_h_0 = delta_ht_0
d_h0 = delta_t0 + delta_h_0
d_c0 = d_h0.dot(O_0).dot(1 - np.square(np.tanh(c_0))) + d_c1.dot(F_1)
d_a0 = d_c0.dot(g_0).dot(1 - np.square(a_0))
d_g0 = d_c0.dot(a_0).dot(g_0).dot(1 - g_0)
d_F0 = d_c0.dot(ct_minus_1).dot(F_0).dot(1 - F_0)
d_O0 = d_h0.dot(np.tanh(c_0)).dot(O_0).dot(1 - O_0)

d_outputs_0 = np.vstack((d_a0, d_g0, d_F0, d_O0))

d_x0 = Wx.T @ (d_outputs_0)
delta_ht_minus_1 = Wh.T @ (d_outputs_0)

# Gradient update of the respective weights and biases after completing
back propagation
d_Wx = np.add(d_outputs_0.dot(x0.reshape(1, 2)),
d_outputs_1.dot(x1.reshape(1, 2)))
d_Wh = d_outputs_1 @ (h_0)
d_b = np.add(d_outputs_0, d_outputs_1)

#Stochastic Gradient Descent with learning rate 0.1
Wx_new = Wx - 0.1 * d_Wx
Wh_new = Wh - 0.1 * d_Wh
b_new = b - 0.1 * d_b
```