Long Short-Term Memory Neural Networks

Aditya N. Govardhan, Lekhavarshini Selvam, Ronik Dhakar

Abstract: Sequential data like sentences or videos represent how humans interpret information. It also indicates how events varies over time such as weather and stock data. Thus prediction and generation of sequential data forms an important problem of machine learning domain. In this report, we introduce the drawbacks of feedforward neural networks in learning sequential data. Recurrent neural networks (RNN) include a memory element in the feedforward structure that leverages the sequential nature of data. However, these memory based networks have technical difficulties in learning long sequential data and fail for practical sizes of data. Long short-term memory (LSTM) neural networks alleviate these technical difficulties using an improved memory element. This report focuses on these shortcomings of RNNs and how LSTMs overcome these shortcomings. It explains in detail how the LSTM memory element (LSTM cell) works and how it mitigates the vanishing gradient problem in backpropagation through time (BPTT). The report also briefs on a demonstrative example that uses LSTM neural network to predict stock market price.

***Index Terms*—sequential data, RNNs, BPTT, vanishing gradient, LSTM cell**

I. Introduction

LSTM neural networks is an improvement on RNNs in terms of its cell structure and its ability to learn long sequences by addressing the issue of vanishing gradients. Thus LSTM neural networks are applied in various machine learning tasks like text generation, handwriting recognition and generation, music generation, image and video captioning, language translation and sentiment analysis.

This report explains the architecture and working of LSTM neural networks and how it addresses the issues with RNNs. The concept is demonstrated with an example of forecasting for stock market data. Since stock market data is sequential in nature, it leverages the power of LSTM neural networks.

II. Background

A. Sequential Data and Drawbacks in Feedforward Neural Networks

Sequential data is represented by $(x^{(1)}, x^{(2)}, ..., x^{(T)})$ where each data point $x^{(t)}$ is a real valued vector. Sequential data can be an input and/or the output of a neural network. For example, for an image captioning problem, the input would be a fixed sized image data and output would be a sequential caption data. For a sentiment analysis problem, input would a sequential opinion data and output would be a positive/negative sentiment. For a language translation problem, both input and output data would be sequential data.

Aditya N. Govardhan, Lekhavarshini Selvam and Ronik Dhakar is with the Arizona State University, Tempe, AZ 85281 USA (e-mail: Aditya.Govardhan@asu.edu, lnolast3@asu.edu, rdhakar@asu.edu).

Feedforward neural networks are designed to take an input of fixed size, process it with fixed size operations and output is of binary or fixed size nature. Although it is possible to carry out sequential data tasks with fixed dimensions of neural networks, it has following drawbacks.

Fixed size models can't model long term dependencies. For example, if we try to model to guess the following blank: "In France, I had a great time and I learnt some of the ___ language", and we have kept the input of length five, then it won't be able to predict the correct answer, i.e., "French" which is dependent on word "France".

Even if we keep a count on frequency of words occurring in a sequence, we lose the context. For example, the following sentences "the food was good, not bad at all" and "the food was bad, not good at all" have the same frequency of words, yet the meaning conveyed is different.

Also, if we use a large size of vector for input, we won't effectively learn the context of the sentence. For example, if the feedforward network learns that "The morning is..." appears at the start of the sentence every time, then it might mispredict the sequence when it appears elsewhere, for example, in the end: "...in the morning."

This motivates us: (1) to deal with variable length sequences, (2) to maintain sequence order, (3) to keep track of long term dependencies and (4) to learn data independent of sequential positions. In light of this, RNNs are used to take advantage of it's memory element.

B. RNNs and its Drawbacks

At an overview, RNNs are structurally similar to feedforward networks. However, the output state of each neuron (unit) is a function of the input as well as its previous state (refer Fig. 1., the subscripts denote the time instant). Fig. 1(a). and 1(b). describes the functioning of memory element. The inputs x_0 and x_1 are applied to the

unit with weight W, which remains constant over time. Also, previous state s_1 is fed back with weight U to the unit to obtain the output state s_1. Thus for Fig. 1.(b).:

$$s_2 = tanh(Wx_1 + Us_1) \quad \text{...(Eq. 1.)}$$

Similar argument can be applied to unit's state at time instant t = 1.

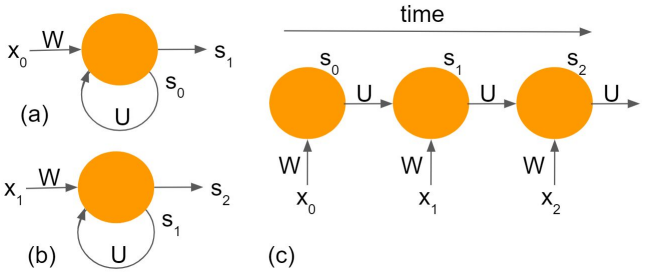

Fig. 1. (a) RNN unit at t = 0 (b) RNN unit at t = 1 (c) RNN "unfolded" through time

For better visualization of operation of memory element, the unit is unfolded through time (Fig. 1(c).). Notice that we use the same parameters W and U at all time instants. And according to Eq. 1., unit's state at time n, s_n, contains information about all previous states, s_0, s_1, s_2, …, s_{n-1}.

RNNs are trained using backpropagation through time (BPTT). In backpropagation, we (1) take the derivative (gradient) of the loss function with respect to each parameter and (2) shift parameters in the opposite direction in order to minimize loss.

In RNN, we have loss at each timestep since we are making a prediction at each timestep. According to Fig. 2., y_1, y_2, …, y_n, are the output at time steps t = 1, 2, …, n and the corresponding losses are J_1, J_2, …, J_n. The loss at each timestep is:

$$loss\ at\ time\ t = J_t(P) \quad \text{...(Eq. 2.)}$$

Where P is network parameter like weights. In BPTT, total loss is the sum of loss at each timestep:

$$total\ loss = J(P) = \sum_t J_t(P) \quad \text{...(Eq. 3.)}$$

Similarly, the total gradient is the sum of gradients across time for each parameter:

$$\frac{\partial J}{\partial P} = \sum_t \frac{\partial J_t}{\partial P} \quad \text{...(Eq. 4.)}$$

Consider the parameter P to be the input weights W. Thus according to chain rule, the gradient at timestep t = 2 would be:

$$\frac{\partial J_2}{\partial W} = \frac{\partial J_2}{\partial y_2}\frac{\partial y_2}{\partial s_2}\frac{\partial s_2}{\partial W} \quad \text{...(Eq. 5.)}$$

From Eq. 1., s_1 also depends on W so $\frac{\partial s_2}{\partial W}$ can't be treated as constant. Thus, it can be seen from Fig. 2. that s_n depends upon s_1, s_2, …, s_{n-1}. Accordingly $\frac{\partial s_2}{\partial W}$ has to be expanded in the following way:

$$\frac{\partial s_2}{\partial s_2}\frac{\partial s_2}{\partial W} + \frac{\partial s_2}{\partial s_1}\frac{\partial s_1}{\partial W} + \frac{\partial s_2}{\partial s_0}\frac{\partial s_0}{\partial W}$$

Thus the gradient for BPTT at timestep t = 2 becomes:

$$\frac{\partial J_2}{\partial W} = \sum_{k=0}^{2} \frac{\partial J_2}{\partial y_2}\frac{\partial y_2}{\partial s_2}\frac{\partial s_2}{\partial s_k}\frac{\partial s_k}{\partial W} \quad \text{...(Eq. 6.)}$$

Similar equation can be obtained for timestep t = n. Last two terms indicate the contributions of W in previous timesteps to the present timestep. This is key to how long term dependencies are modelled. The parameters are shifted in opposite directions in order to minimize loss and thus BPTT is carried out. However, RNNs are difficult to train in real scenarios. It can be explained in the following way.

It can be noticed in Eq. 6. that chain rule can be applied to the term $\frac{\partial s_2}{\partial s_k}$ leading to $\frac{\partial s_2}{\partial s_1}\frac{\partial s_1}{\partial s_0}$ at k = 0. Similarly, for timestep t = n, $\frac{\partial s_n}{\partial s_0}$ can be expanded as $\frac{\partial s_n}{\partial s_{n-1}}\frac{\partial s_{n-1}}{\partial s_{n-2}}\cdots\frac{\partial s_3}{\partial s_2}\frac{\partial s_2}{\partial s_1}\frac{\partial s_1}{\partial s_0}$. As the gap between timesteps get bigger, number of terms in this product increases. Theoretically, the term $\frac{\partial s_n}{\partial s_{n-1}}$ depends upon matrix W (which is generally sampled from a standard normal distribution) and derivative of the activation function f' (which is generally sigmoid or tansigmoid). W and f' are mostly less than one. Thus, we are multiplying a lot of small numbers together. Thus the product $\frac{\partial s_n}{\partial s_{n-1}}\frac{\partial s_{n-1}}{\partial s_{n-2}}\cdots\frac{\partial s_3}{\partial s_2}\frac{\partial s_2}{\partial s_1}\frac{\partial s_1}{\partial s_0}$ is a multiplication of increasingly smaller numbers and from the generalized form of Eq. 6., the errors due to past timesteps have increasingly smaller gradients. This is known as vanishing gradient problem. Vanishing gradients leads to the parameters being biased to capture short-term dependencies. The errors that arise from long-term dependencies find it harder to propagate to future timesteps.

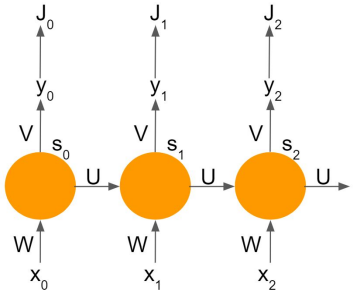

Fig. 2. Time unfolded RNN with outputs (y) and losses (J)

It is also possible that the weight matrix W is greater than one, which intuitively leads to exploding gradients problem. However, this problem can be mitigated by clipping the gradient if it reaches a certain threshold (gradient clipping).

To mitigate the issue of vanishing gradients, multiple methods can be used. A more suitable activation function with derivative greater than one (ReLU) can be chosen. Also, one more suitable method would be initialization of weights with identity matrix and biases with zero. However, a more innovative solution is obtained by modifying the cell structure of the RNN cell. This leads to the architecture of Long Short-Term Memory (LSTM) neural networks, which is explained in the following sections.

III. LSTM Neural Networks

LSTM neural networks were proposed by Sepp Hochreiter and Jürgen Schmidhuber in 1995. LSTM neural network architecture varies at the cell structure level. LSTM cell was designed to mitigate the exploding and vanishing gradient problem. From Fig. 3. it can be seen that LSTM cell has one more output of cell state which is in parallel to the standard RNN cell state output.

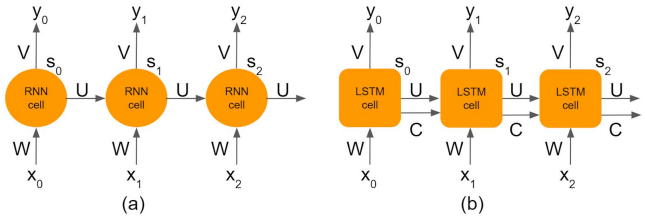

Fig. 3. (a) RNN structure (b) LSTM network structure

In a standard RNN, repeating modules contain a simple computation node (i.e. sigmoid or tansigmoid). However, LSTM repeating modules contain interacting layers that control information flow. LSTM cell has a three step process: 1) *Forget:* Forget irrelevant parts of the previous state; for example, if a sentence is being modelled and a new subject appears, there might be a requirement to forget things about the previous subject since the following words would be irrelevant to previous subject 2) *Update:* the next step is an update step; for example, if a new subject occurs, then at this step the cell state is updated with the new information about the subject 3) *Output:* output step is used to produce the relevant information about the new subject. Each of these three steps is implemented using a set of logic gates, and the logic gates are implemented using sigmoid functions.

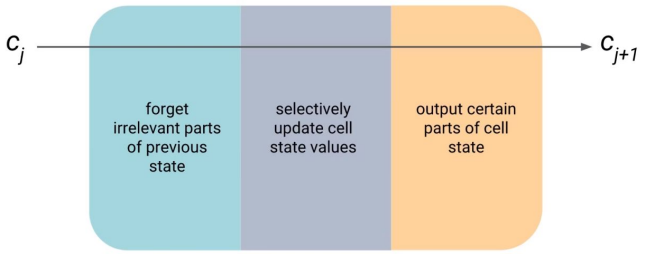

Fig. 4. LSTM cell functioning

Fig. 5. shows the internal structure of an LSTM cell. LSTM cells maintain a cell state c_t where it's easy for information to flow. This cell state runs throughout the chain of repeating modules and only a couple of linear interactions, pointwise multiplication and addition, update the value of c_t.

Information is added or removed to the inputs of the cell through structures called gates. These gates consist of sigmoid layer followed by a pointwise multiplication. Due to the nature of sigmoid function, the input is forgotten if the sigmoid layer output is zero and remembered if the sigmoid layer output is one.

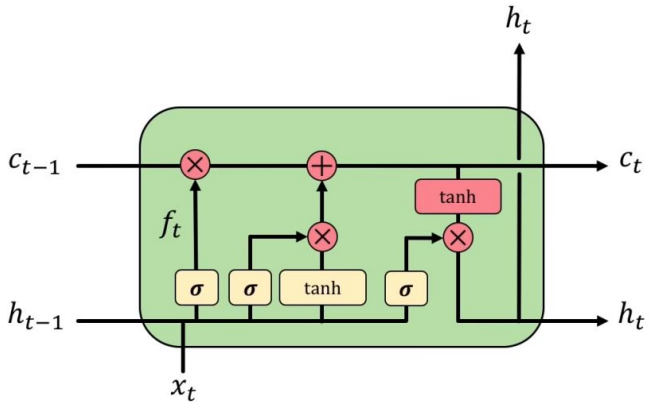

Fig. 5. Internal structure of LSTM cell

A) Forget step:

LSTM forgets irrelevant parts of the previous state using the gate f_t. The forget gate f_t is parameterized using a set of weights of biases like a neural network layer parameterized by the following equation:

$$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f) \quad \text{...(Eq. 7.)}$$

This gate looks at the previous state h_{t-1} and input x_t (see Fig. 6.) and outputs a value between zero and one, where zero represents completely forgetting previous information and one represents completely retaining previous information.

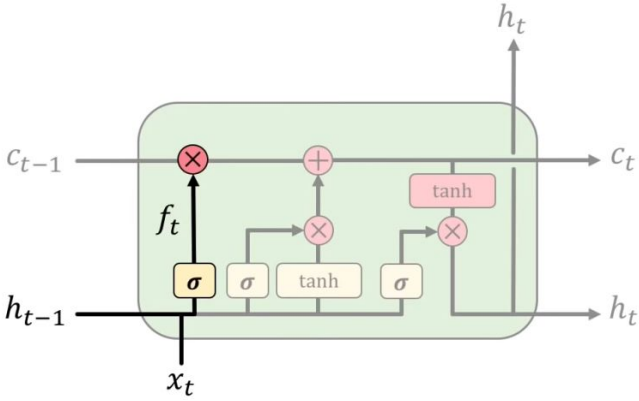

Fig. 6. Forget step

B) Update step

The update step is carried out in two substeps. In the first substep (see Fig. 7.), the sigmoid layer decides what values need to be updated parameterized by the following equation:

$$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i) \quad \text{...(Eq. 8.)}$$

The tansigmoid layer generates new vector of "candidate values" that could be added to the cell state. It is parameterized by the following equation:

$$\overline{C}_t = tanh(W_C[h_{t-1}, x_t] + b_C) \quad \text{...(Eq. 9.)}$$

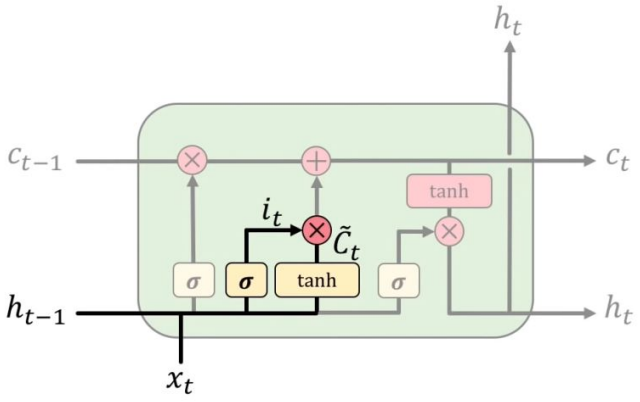

Fig. 7. Update substep I

In the second substep (see Fig. 8.), we apply forget operation to the old state c_{t-1} and add new candidate $\overline{C}_t$ to it, scaled by how much we want to update it, i.e. i_t. This is modelled by the following equation:

$$c_t = f_t * c_{t-1} + i_t * \overline{C}_t \quad \text{…(Eq. 10.)}$$

It can be seen that the amount of old information and new information to be kept is controlled by sigmoid outputs f_t and i_t, while tansigmoid layer generates new candidate $\overline{C}_t$.

C) Output step

Finally, the output h_t is a filtered version of newly computed c_t (see Fig. 9.). A sigmoid layer decides what parts of the previous state to output, parameterized by the following equation:

$$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o) \quad \text{…(Eq. 11.)}$$

The newly computed cell state c_t is passed through a tansigmoid layer and multiplied with the output of the sigmoid layer from Eq.11. as shown:

$$h_t = o_t * tanh(c_t) \quad \text{…(Eq. 12.)}$$

Essentially this amounts to transforming or filtering the cell state using the tansigmoid layer and the sigmoid layer.

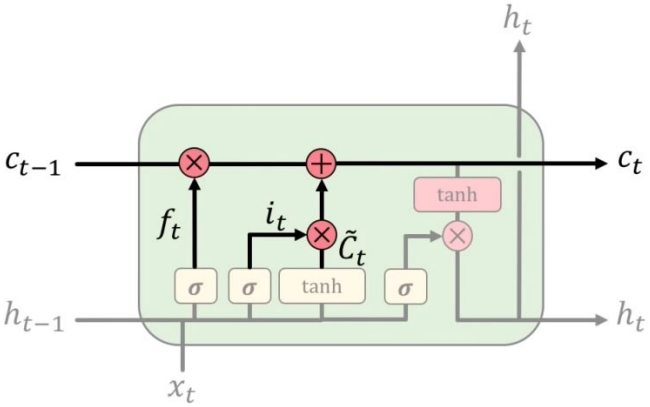

Fig. 8. Update substep II

To get an idea of how LSTM cell mitigates the vanishing gradient problem, consider the derivative of c_t with respect to c_{t-1} from Eq. 10:

$$\frac{c_t}{c_{t-1}} \approx \sigma(W_f[h_{t-1}, x_t] + b_f) \quad \text{…(Eq. 13.)}$$

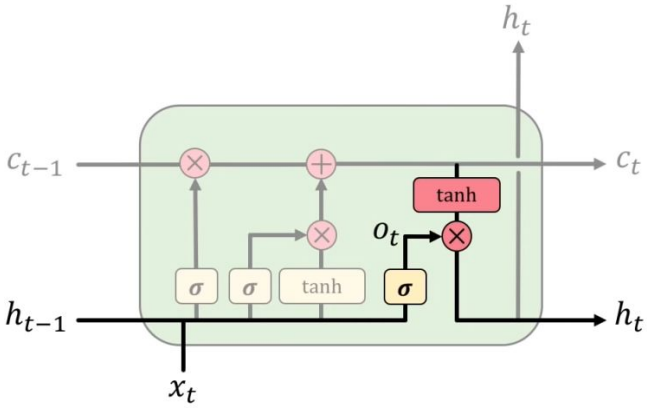

Fig. 9. Output step

This is assuming that the $tanh(W_C[h_{t-1}, x_t] + b_C)$ term is zero when taken a derivative with respect to c_{t-1}. Thus, it can be seen that the gradient doesn't vanish but is determined by the sigmoid function in Eq. 13. Thus LSTM maintains a separate cell state from what is outputted and uses gate control mechanism to control the flow of information as well as mitigates vanishing gradient problem. To demonstrate the usefulness of LSTM neural networks, a demonstration for stock price prediction is presented.

IV. Stock Price Prediction Using LSTM

The dataset used for prediction contains information about the stock prices of last 3 years of yahoo finance and the aim is to predict the closing price of the stock. LSTM is used for this purpose as it can retain information from the past and the closing prices are predicted based on the previous closing prices of the stock.

```
#Weights for the input gate
weights_input_gate = tf.Variable(tf.truncated_normal([1, h], stddev=0.05))
weights_input_hidden = tf.Variable(tf.truncated_normal([h, h], stddev=0.05))
bias_input = tf.Variable(tf.zeros([h]))

#weights for the forgot gate
weights_forget_gate = tf.Variable(tf.truncated_normal([1, h], stddev=0.05))
weights_forget_hidden = tf.Variable(tf.truncated_normal([h, h], stddev=0.05))
bias_forget = tf.Variable(tf.zeros([h]))

#weights for the output gate
weights_output_gate = tf.Variable(tf.truncated_normal([1, h], stddev=0.05))
weights_output_hidden = tf.Variable(tf.truncated_normal([h, h], stddev=0.05))
bias_output = tf.Variable(tf.zeros([h]))

#weights for the memory cell
weights_memory_cell = tf.Variable(tf.truncated_normal([1, h], stddev=0.05))
weights_memory_cell_hidden = tf.Variable(tf.truncated_normal([h, h], stddev=0.05))
bias_memory_cell = tf.Variable(tf.zeros([h]))

#Output layer weigts
weights_output = tf.Variable(tf.truncated_normal([h, 1], stddev=0.05))
bias_output_layer = tf.Variable(tf.zeros([1]))
```

Fig. 10. Weight declaration of gates

Initially the closing price is scaled between -1 and 1 using min-max scaler in order to bring stationarity in data. The data is split into training and testing data with 80% data as training and 20% data as testing. There are 5 input parameters namely (1) window size (w) for deciding the number of inputs, (2) hidden layer (h), (3) batch size (b) for deciding the number of windows to be given at a time, (4) clip margin for deciding the threshold value in order to

avoid gradient explosion and is essential as it involves matrix multiplication and (5) epochs for deciding number of iterations during forward and backward movement in network and learning rate for making loss function reach the optimal value.

LSTM consist of gates namely input, forget, output and cell state and Fig. 10. shows assigning of weight values to different gates which would be multiplied with inputs and previous state output to decide the important information to preserve. The equation of forget gate, input gate, output gate and cell state as mentioned earlier has been used for calculating the value of the gates.

```
Epoch 0/200  Current loss: 0.09508194029331207
Epoch 40/200  Current loss: 0.01598675735294819
Epoch 80/200  Current loss: 0.015596196986734867
Epoch 120/200  Current loss: 0.014485039748251438
Epoch 160/200  Current loss: 0.013005262240767479
```

Fig. 11. Loss for every 40 epochs

The number of iterations or epochs are carried out during each input window and Fig. 11. shows the loss after every 40 epochs and it is evident that the losses keep on decreasing after every 40 epochs making the model to learn better and fit the original data accurately.

```
losses = []

for i in range(len(outputs)):
    losses.append(tf.losses.mean_squared_error(tf.reshape(targets[i], (-1, 1)),
                                               outputs[i]))

loss = tf.reduce_mean(losses)
```

Fig. 12. Mean Squared Error calculation for every output

The mean square loss is calculated for each output so as to minimize the error between the actual and the predicted output so as to adjust weights making the loss function reach the optimal value. Fig. 12. shows the code for calculating the loss.

```
outputs = []
for i in range(b):

    batch_state = np.zeros([1, h], dtype=np.float32)
    batch_output = np.zeros([1, h], dtype=np.float32)

    for j in range(w):
        batch_state, batch_output = LSTM_cell(tf.reshape(inputs[i][j], (-1, 1)),
                                              batch_state, batch_output)

    outputs.append(tf.matmul(batch_output, weights_output) + bias_output_layer)
outputs
```

Fig. 13. Predicting Output

Fig. 13. shows the input given in batches of window size in order to get the next output and the final output is predicted using output weights and batch output Initially the batch state and the batch output is kept zero and each prediction is stored in list named outputs.

Fig. 14. shows the output graph where blue line represents the original data, red line represents training data and green line represents testing data. It can be seen that the model has been trained quite accurately as compared to the original data and also the plot of testing data corresponding to original output shows the proximity of testing data compared to original data.

Fig. 14. Original data, Training result and Testing result

V. Conclusion

Eq. 13. indicates that the gradient of BPTT for LSTM cell is dependent on sigmoid function rather than a series of products. This helps in selectively determining if a particular dependency needs to be learnt or not, irrelevant of how far the dependency is from the present data. As it is clear from Fig. 14.. the demonstration also indicates the usefulness of LSTM neural networks in modelling sequential data.

Given these qualities of LSTM neural networks, they find a variety of applications in sequential data prediction and generation.

References

[1] Sherstinsky, A. (2018). Fundamentals of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) Network. CoRR, abs/1808.03314.

[2] Lipton, Z.C. (2015). A Critical Review of Recurrent Neural Networks for Sequence Learning. CoRR, abs/1506.00019.

[3] Gers, F.A., Schmidhuber, J., & Cummins, F.A. (2000). Learning to Forget: Continual Prediction with LSTM. Neural Computation, 12, 2451-2471.

[4] Pascanu, R., Mikolov, T., & Bengio, Y. (2013). On the difficulty of training recurrent neural networks. ICML.

[5] Józefowicz, R., Zaremba, W., & Sutskever, I. (2015). An Empirical Exploration of Recurrent Network Architectures. ICML.

[6] Hochreiter, S., & Schmidhuber, J. (1997). Long Short-Term Memory. Neural Computation, 9, 1735-1780

[7]https://heartbeat.fritz.ai/a-beginners-guide-to-implementing-long-short-term-memory-networks-lstm-eb7a2ff09a27 (Source Code Reference)

[8] MIT's course on deep learning - 6.S191 Introduction to deep learning