Long Short-Term Memory (LSTM) Networks

Table of Contents

1. Introduction to LSTM

LSTM (Long Short-Term Memory) networks are a special kind of recurrent neural network (RNN) capable of learning long-term dependencies. They were introduced to overcome the limitations of standard RNNs, such as short-term memory and vanishing gradient problems. LSTMs are widely used in tasks like speech recognition, time-series prediction, and natural language processing.

2. Issues with Recurrent Neural Networks (RNNs)

Short-term Memory Problem

Traditional RNNs struggle to remember information from distant time steps. They primarily rely on short-term memory, making them ineffective for tasks requiring long-term dependencies.

Vanishing and Exploding Gradient Problems

During backpropagation, gradients can become excessively small (vanishing gradient) or excessively large (exploding gradient), leading to slow learning or instability.

Difficulty in Learning Long-term Dependencies

Standard RNNs find it hard to retain useful information across long sequences, impacting performance in tasks that require context retention over time.

3. LSTM Networks: Overcoming RNN Limitations

LSTMs address the issues of standard RNNs by introducing ***memory cells*** and ***gate mechanisms***. Instead of relying solely on short-term memory, LSTMs maintain a more stable long-term memory, improving performance in sequential data tasks. Unlike RNN cells, LSTM cells selectively decide what to store, discard, and output, making them more efficient.

4. LSTM Architecture

LSTMs consist of three key components:

1. **Memory cell (C_t):** Stores long-term information.
2. **Gates (Forget, Input, Output):** Regulate information flow.
3. **Hidden state (h_t):** Used for predictions and next time-step computations.

The memory cell and gates work together to update and manage information efficiently.

5. LSTM Structure: The Three Gates

Forget Gate

- Determines which information from the previous cell state should be discarded.
- Uses the sigmoid function (σ) to generate values between 0 and 1.
- Formula:

Input Gate

- Decides which new information should be added to memory.
- Uses a combination of the sigmoid (σ) and tanh functions.
- Formula:

Output Gate

- Controls what part of the memory should be output at the current time step.

- Uses the sigmoid function to determine which information to pass forward.
- Formula:

6. LSTM Cell State and Updating Mechanism

Candidate State Computation

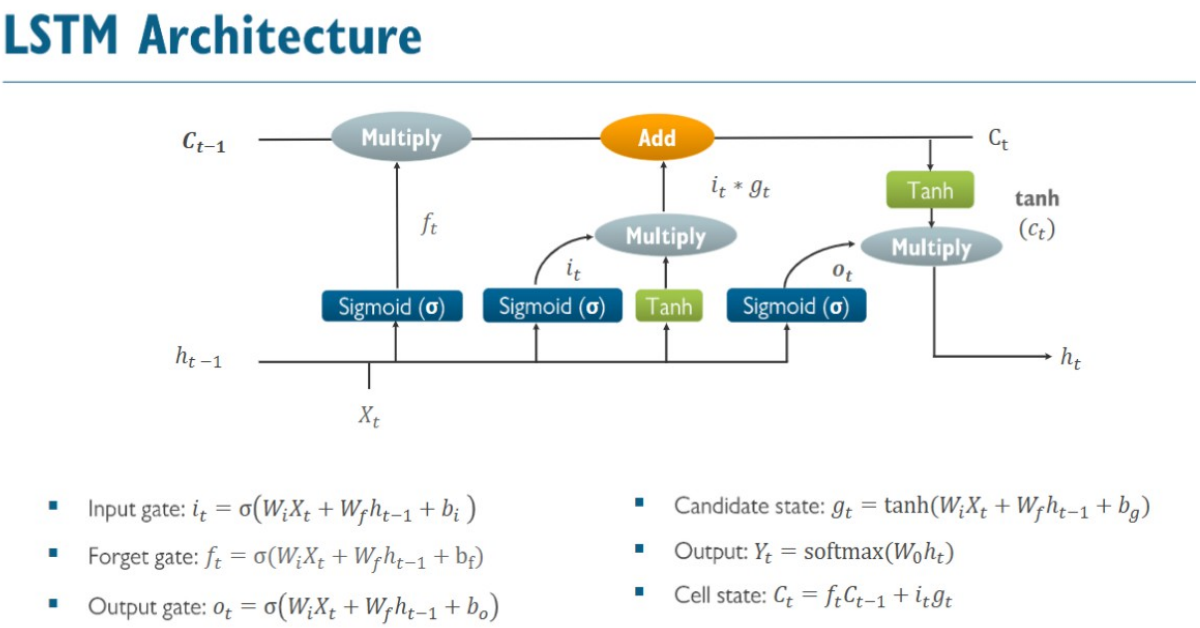
- The candidate memory state is computed using the tanh function.
- Formula:

Memory Update

- The cell state is updated based on the forget and input gate values.
- Formula:
- The tanh activation function ensures values remain between -1 and +1, stabilizing learning.

7. Summary and Key Takeaways

- RNNs suffer from short-term memory and vanishing gradients.
- LSTMs introduce ***memory cells*** and ***gates*** to store long-term dependencies.
- ***Forget, Input, and Output gates*** control the information flow in LSTMs.
- The ***memory cell*** maintains important information, making LSTMs effective for sequence-based tasks.
- LSTMs are widely used in ***NLP, speech processing, and time-series forecasting***.



- Input gate: $i_t = \sigma(W_i X_t + W_f h_{t-1} + b_i)$
- Forget gate: $f_t = \sigma(W_i X_t + W_f h_{t-1} + b_f)$
- Output gate: $o_t = \sigma(W_i X_t + W_f h_{t-1} + b_o)$
- Candidate state: $g_t = \tanh(W_i X_t + W_f h_{t-1} + b_g)$
- Output: $Y_t = \text{softmax}(W_0 h_t)$
- Cell state: $C_t = f_t C_{t-1} + i_t g_t$

8. Annex

Common Activation Functions and Their Formulas

Here are the formulas for commonly used activation functions in neural networks:

1. **Sigmoid Function**

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

   Output range: (0,1)
   - Used in LSTMs for gates.
2. **Tanh (Hyperbolic Tangent) Function**

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

   Output range: (−1,1)
   - Used in LSTMs for candidate cell state.
3. **ReLU (Rectified Linear Unit)**

$$ReLU(x) = \max(0, x)$$

   - Output range: (0,∞)
   - Used in deep networks to prevent vanishing gradients.
4. **Softmax Function** (for multi-class classification)

$$S(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

   - Converts logits into probability distributions.

key formulas used in LSTM networks:

Forget Gate

Determines which information from the previous cell state should be discarded.

$$f_t = \sigma(W_f X_t + U_f h_{t-1} + b_f)$$

Input Gate

Decides which new information should be added to memory.

$$i_t = \sigma(W_i X_t + U_i h_{t-1} + b_i)$$

Candidate Cell State

$$g_t = \tanh(W_g X_t + U_g h_{t-1} + b_g)$$

Cell State Update

The cell state is updated based on the forget and input gate values.

$$C_t = f_t C_{t-1} + i_t g_t$$

Output Gate

Controls what part of the memory should be output at the current time step.

$$o_t = \sigma(W_o X_t + U_o h_{t-1} + b_o)$$

Hidden State Update

The hidden state is computed using the updated cell state and the output gate.

$$h_t = o_t \tanh(C_t)$$

These formulas define how LSTM networks manage long-term dependencies effectively.

Example: Calculating Weight and Bias Matrices in LSTMs

Let's assume we have an LSTM cell with:

- Input size = 3 (features in input Xt)
- Hidden size = 2 (number of LSTM units)

The weight matrices are:

1. **Weights for input X_t**
   W_i, W_f, W_o, W_g → Shape: $(2, 3)$ (hidden size × input size)
2. **Weights for hidden state h_{t-1}**
   U_i, U_f, U_o, U_g → Shape: $(2, 2)$ (hidden size × hidden size)
3. **Bias vectors**
   b_i, b_f, b_o, b_g → Shape: $(2, 1)$ (hidden size × 1)

Numerical Example

Let's consider a simple input:

Let's consider a simple input:

$$X_t = \begin{bmatrix} 0.5 \\ 0.1 \\ -0.3 \end{bmatrix}$$

Randomly initialized weights and biases:

$$W_i = \begin{bmatrix} 0.2 & 0.4 & -0.5 \\ 0.1 & -0.3 & 0.7 \end{bmatrix}, \quad U_i = \begin{bmatrix} 0.5 & -0.2 \\ -0.1 & 0.4 \end{bmatrix}$$

$$h_{t-1} = \begin{bmatrix} 0.2 \\ -0.1 \end{bmatrix}, \quad b_i = \begin{bmatrix} 0.1 \\ -0.2 \end{bmatrix}$$

Now, calculating the input gate activation:

$$i_t = \sigma(W_i X_t + U_i h_{t-1} + b_i)$$

Breaking it down:

$$W_i X_t = \begin{bmatrix} 0.2 & 0.4 & -0.5 \\ 0.1 & -0.3 & 0.7 \end{bmatrix} \begin{bmatrix} 0.5 \\ 0.1 \\ -0.3 \end{bmatrix} = \begin{bmatrix} 0.2(0.5) + 0.4(0.1) + (-0.5)(-0.3) \\ 0.1(0.5) + (-0.3)(0.1) + 0.7(-0.3) \end{bmatrix}$$

$$= \begin{bmatrix} 0.1 + 0.04 + 0.15 \\ 0.05 - 0.03 - 0.21 \end{bmatrix} = \begin{bmatrix} 0.29 \\ -0.19 \end{bmatrix}$$

$$U_i h_{t-1} = \begin{bmatrix} 0.5 & -0.2 \\ -0.1 & 0.4 \end{bmatrix} \begin{bmatrix} 0.2 \\ -0.1 \end{bmatrix} = \begin{bmatrix} 0.5(0.2) + (-0.2)(-0.1) \\ -0.1(0.2) + 0.4(-0.1) \end{bmatrix}$$

$$= \begin{bmatrix} 0.1 + 0.02 \\ -0.02 - 0.04 \end{bmatrix} = \begin{bmatrix} 0.12 \\ -0.06 \end{bmatrix}$$

$$i_t = \sigma(\begin{bmatrix} 0.29 \\ -0.19 \end{bmatrix} + \begin{bmatrix} 0.12 \\ -0.06 \end{bmatrix} + \begin{bmatrix} 0.1 \\ -0.2 \end{bmatrix})$$

$$= \sigma(\begin{bmatrix} 0.51 \\ -0.45 \end{bmatrix})$$

Applying the **sigmoid function**:

$$\sigma(0.51) = \frac{1}{1 + e^{-0.51}} \approx 0.625$$

$$\sigma(-0.45) = \frac{1}{1 + e^{0.45}} \approx 0.389$$

Final result:

$$i_t = \begin{bmatrix} 0.625 \\ 0.389 \end{bmatrix}$$

Jupyter Notebook Demo

I'll now generate a **Jupyter Notebook** that contains:

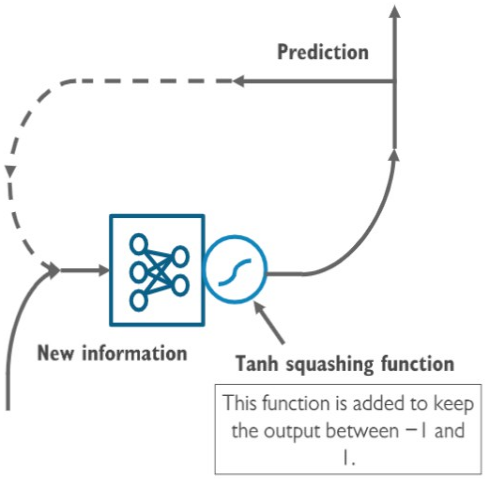
- Implementation of LSTM step-by-step.
- Calculation of weight and bias transformations.
- Visualization of activation functions.

```
import numpy as np
import matplotlib.pyplot as plt

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

```
def tanh(x):
    return np.tanh(x)

# Sample input vector X_t (3 features)
X_t = np.array([[0.5], [0.1], [-0.3]])

# Sample hidden state from previous timestep h_{t-1} (2 units)
h_t_1 = np.array([[0.2], [-0.1]])

# Random weight matrices for input and hidden layers
W_i = np.array([[0.2, 0.4, -0.5], [0.1, -0.3, 0.7]])
U_i = np.array([[0.5, -0.2], [-0.1, 0.4]])
b_i = np.array([[0.1], [-0.2]])

# Compute input gate activation
z_i = np.dot(W_i, X_t) + np.dot(U_i, h_t_1) + b_i
i_t = sigmoid(z_i)

print("Input Gate Activation:")
print(i_t)

# Generate plots for activation functions
x_vals = np.linspace(-5, 5, 100)
plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
plt.plot(x_vals, sigmoid(x_vals))
plt.title("Sigmoid Function")
plt.grid()

plt.subplot(1, 2, 2)
plt.plot(x_vals, tanh(x_vals))
plt.title("Tanh Function")
plt.grid()

plt.show()
```

Issues with RNN

Each prediction looks back one time step as it has very short-term memory; therefore, it does not use the information from further back thereby failing to connect the information.



Resolving Issues with RNN: Memory

- Memory helps to remember what happened many time stamps ago.
- This is where LSTM comes into the picture.