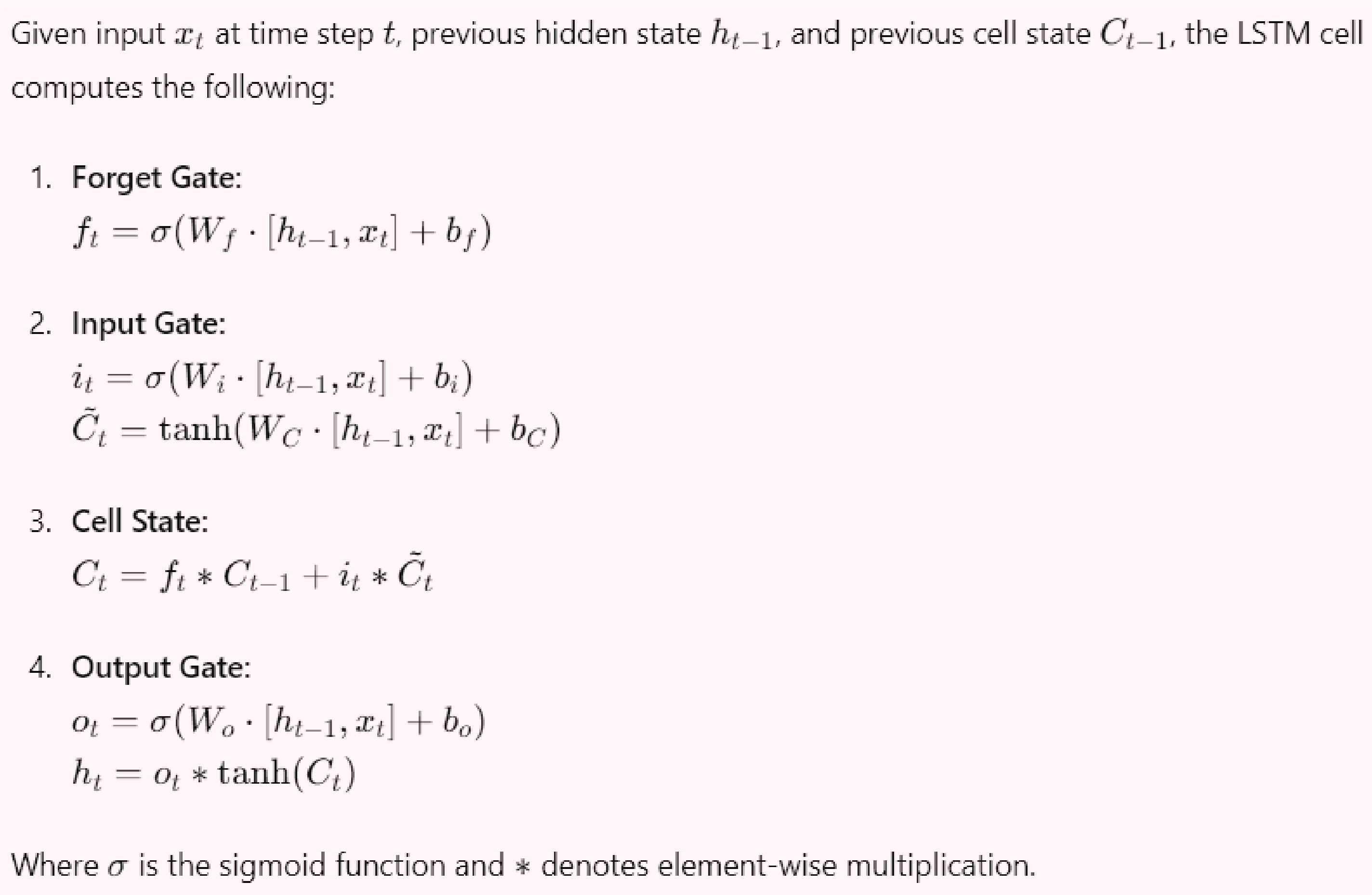
Implementing a Long Short-Term Memory from Scratch in Python

Table of Contents

1. Introduction to Long Short-Term Memory

Long Short-Term Memory (LSTM) networks are a type of Recurrent Neural Network (RNN) capable of learning long-term dependencies. LSTMs are explicitly designed to avoid the long-term dependency problem, making them well-suited for tasks like time series prediction, speech recognition, and text generation. In this guide, we will explore the fundamental concepts of LSTMs and implement an LSTM from scratch in Python without relying on high-level libraries like TensorFlow or PyTorch.

Mathematical Formulation



Given input x_t at time step t, previous hidden state h_{t-1}, and previous cell state C_{t-1}, the LSTM cell computes the following:

1. **Forget Gate:**
   $f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$

2. **Input Gate:**
   $i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$
   $\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$

3. **Cell State:**
   $C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$

4. **Output Gate:**
   $o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$
   $h_t = o_t * \tanh(C_t)$

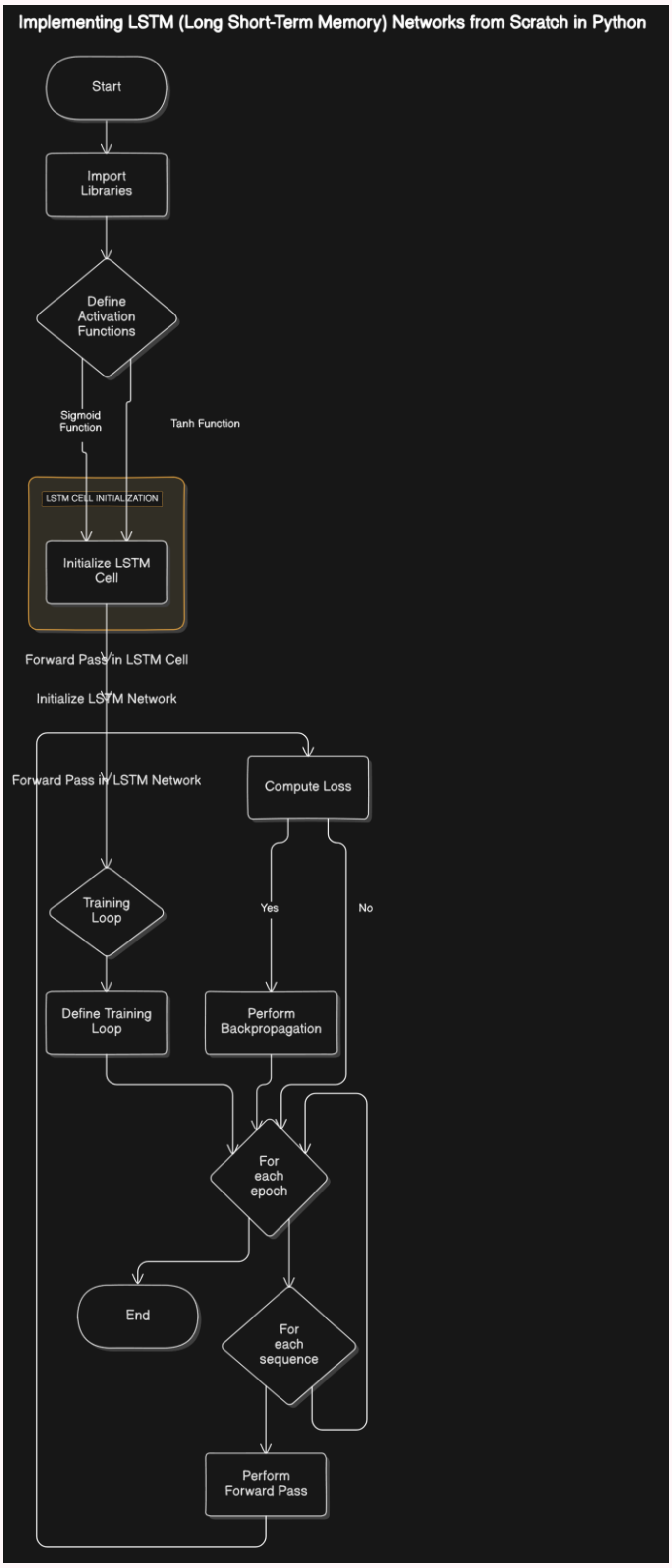
Where σ is the sigmoid function and $*$ denotes element-wise multiplication.

2. The Structure of a LSTMs

LSTM networks are composed of LSTM cells, each of which has three main components:

1. **Forget Gate:** Decides what information to discard from the cell state.
2. **Input Gate:** Decides which values from the input will update the cell state.
3. **Output Gate:** Decides what the next hidden state should be.

Here is a simple diagram to illustrate a basic LSTMs:



3. Implementation in Python

Let's implement a simple Long Short-Term Memory in Python to classify data. We'll use NumPy for numerical computations.

Step 1: Importing Required Libraries

We will start by importing necessary libraries like NumPy for numerical operations.

```
import numpy as np
```

Step 2: Sigmoid and Tanh Activation Functions

Define the activation functions sigmoid and tanh.

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

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

Step 3: LSTM Cell Class

Create an LSTM cell class that encapsulates the forward pass logic.

```
class LSTMCell:
    def __init__(self, input_dim, hidden_dim):
        self.input_dim = input_dim
        self.hidden_dim = hidden_dim

        # Initialize weights
        self.W_f = np.random.randn(hidden_dim, hidden_dim + input_dim)
        self.b_f = np.zeros((hidden_dim, 1))
        self.W_i = np.random.randn(hidden_dim, hidden_dim + input_dim)
        self.b_i = np.zeros((hidden_dim, 1))
        self.W_C = np.random.randn(hidden_dim, hidden_dim + input_dim)
        self.b_C = np.zeros((hidden_dim, 1))
        self.W_o = np.random.randn(hidden_dim, hidden_dim + input_dim)
        self.b_o = np.zeros((hidden_dim, 1))

    def forward(self, x_t, h_prev, C_prev):
        # Concatenate h_prev and x_t
        concat = np.vstack((h_prev, x_t))

        # Forget gate
        f_t = sigmoid(np.dot(self.W_f, concat) + self.b_f)

        # Input gate
        i_t = sigmoid(np.dot(self.W_i, concat) + self.b_i)
        C_tilde = tanh(np.dot(self.W_C, concat) + self.b_C)

        # Cell state
        C_t = f_t * C_prev + i_t * C_tilde

        # Output gate
        o_t = sigmoid(np.dot(self.W_o, concat) + self.b_o)
        h_t = o_t * tanh(C_t)

        return h_t, C_t
```

Step 4: LSTM Network Class

Create an LSTM network class that handles multiple LSTM cells and processes sequences of data.

```
class LSTMNetwork:
    def __init__(self, input_dim, hidden_dim, output_dim):
        self.hidden_dim = hidden_dim
        self.lstm_cell = LSTMCell(input_dim, hidden_dim)
        self.W_y = np.random.randn(output_dim, hidden_dim)
        self.b_y = np.zeros((output_dim, 1))

    def forward(self, X):
        h_t = np.zeros((self.hidden_dim, 1))
        C_t = np.zeros((self.hidden_dim, 1))
        outputs = []

        for x_t in X:
            h_t, C_t = self.lstm_cell.forward(x_t, h_t, C_t)
            y_t = np.dot(self.W_y, h_t) + self.b_y
            outputs.append(y_t)

        return outputs
```

Step 5: Training the LSTM Network

Define a simple training loop to train the LSTM network using backpropagation through time (BPTT).

```
# Assuming we have a dataset of sequences X and corresponding labels Y
# X shape: (number of sequences, sequence length, input_dim)
# Y shape: (number of sequences, output_dim)

def train_lstm(X, Y, input_dim, hidden_dim, output_dim, learning_rate=0.01, epochs=100):
    lstm_network = LSTMNetwork(input_dim, hidden_dim, output_dim)

    for epoch in range(epochs):
        total_loss = 0

        for i in range(len(X)):
            outputs = lstm_network.forward(X[i])
            loss = np.mean((outputs[-1] - Y[i])**2)
            total_loss += loss

            # Backpropagation and weight updates would go here
            # (Note: Implementing BPTT is complex and beyond the scope of this basic guide)

        print(f'Epoch {epoch+1}/{epochs}, Loss: {total_loss/len(X)}')

    return lstm_network
```

Step 6: Sample Data and Execution

To test the implemented LSTM, you can generate some sample data and run the training function

```
# Sample data generation
np.random.seed(0)
X = [np.random.randn(10, 3) for _ in range(100)]  # 100 sequences, each of length 10, input_dim=3
Y = [np.random.randn(2, 1) for _ in range(100)]   # 100 labels, output_dim=2

# Train the LSTM network
trained_lstm = train_lstm(X, Y, input_dim=3, hidden_dim=5, output_dim=2)
```

Conclusion

Building an LSTM from scratch provides a deeper understanding of how these networks work. This guide covered the fundamental concepts and step-by-step implementation of an LSTM network in Python. While this implementation is simplified and doesn't include backpropagation through time, it serves as a foundation for further exploration and learning.

Constructive comments and feedback are welcomed