Long Short-Term Memory (LSTM) from Scratch

Bhanu Prasanna Koppolu

Table of contents

Source Code

Full implementation available at DL_from_scratch/lstm.py

Long Short-Term Memory (LSTM)

LSTM addresses the vanishing gradient problem of vanilla RNNs by introducing a **cell state** and gating mechanisms.

Initial Conditions

- $h_0 = 0$ (initial hidden state)
- $C_0 = 0$ (initial cell state)

LSTM Gates

The LSTM has four gates, each with its own weights:

1. **Forget Gate** - What to forget from cell state:

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

2. **Input Gate** - What new information to store:

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

3. **Candidate Gate** - New candidate values:

$$\tilde{C}_t = \tanh(W_g \cdot [h_{t-1}, x_t] + b_g)$$

4. **Output Gate** - What to output:

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$

Cell State and Hidden State Update

Cell State Update:

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

Hidden State Update:

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

Output Prediction

$$\hat{y} = W_y \cdot h_T + b_y$$

Implementation

```
class LSTM:
    """LSTM for sequence prediction.

    Full scratch implementation with forget, input, candidate,
    and output gates.
    """

    def __init__(self,
                 input_dim=1,     # D
                 hidden_dim=100,  # H
                 seq_len=50):     # T

        self.output_dim = input_dim  # O

        # Forget Gate: W_f shape (H, H+D), b_f shape (H, 1)
        self.W_f = np.random.randn(hidden_dim, hidden_dim + input_dim) * 0.01
        self.b_f = np.random.randn(hidden_dim, 1) * 0.01

        # Input Gate: W_i shape (H, H+D), b_i shape (H, 1)
        self.W_i = np.random.randn(hidden_dim, hidden_dim + input_dim) * 0.01
        self.b_i = np.random.randn(hidden_dim, 1) * 0.01

        # Candidate Gate: W_g shape (H, H+D), b_g shape (H, 1)
        self.W_g = np.random.randn(hidden_dim, hidden_dim + input_dim) * 0.01
        self.b_g = np.random.randn(hidden_dim, 1) * 0.01

        # Output Gate: W_o shape (H, H+D), b_o shape (H, 1)
        self.W_o = np.random.randn(hidden_dim, hidden_dim + input_dim) * 0.01
        self.b_o = np.random.randn(hidden_dim, 1) * 0.01

        # Output Prediction: W_y shape (O, H), b_y shape (O, 1)
        self.W_y = np.random.randn(self.output_dim, hidden_dim) * 0.01
        self.b_y = np.random.randn(self.output_dim, 1) * 0.01

    def forward(self, x):
        """Forward pass through LSTM.

        For each timestep t:
        1. Concatenate h_{t-1} and x_t to form z_t
        2. Compute forget gate: f_t =  (W_f @ z_t + b_f)
```

```
        3. Compute input gate: i_t =  (W_i @ z_t + b_i)
        4. Compute candidate: g_t = tanh(W_g @ z_t + b_g)
        5. Update cell state: C_t = f_t   C_{t-1} + i_t   g_t
        6. Compute output gate: o_t =  (W_o @ z_t + b_o)
        7. Compute hidden state: h_t = o_t   tanh(C_t)

        Final prediction: y_hat = W_y @ h_T + b_y
        """
        ...

    def backward(self, y_true, y_pred, lr=0.001):
        """Backpropagation through time for LSTM.

        Gradient computation for all gates:

        1. Output layer gradients
        2. For t = T down to 1:
           - G_o = G_h   tanh(C_t)
           - G_C += G_h   o_t   (1 - tanh²(C_t))
           - G_f = G_C   C_{t-1}
           - G_i = G_C   g_t
           - G_g = G_C   i_t
           - Compute sigmoid/tanh derivatives
           - Accumulate weight gradients
           - Propagate G_h and G_C backwards

        3. Clip gradients and update weights
        """
        # Output gradients
        G_wy = (y_pred - y_true) @ self.h[self.seq_len].T  # 1 × H
        G_by = y_pred - y_true                              # 1 × 1

        G_h = self.W_y.T @ (y_pred - y_true)  # H × 1
        G_C = np.zeros((self.hidden_dim, 1))  # H × 1

        # Accumulate gradients for each gate
        G_wf = G_bf = G_wi = G_bi = G_wg = G_bg = G_wo = G_bo = 0

        for t in range(self.seq_len, 0, -1):
            # Gate gradients with sigmoid/tanh derivatives
            ...

            # Accumulate weight gradients
            ...

            # Propagate to previous timestep
            G_z = self.W_f.T @ G_f_pre + self.W_i.T @ G_i_pre + \
                  self.W_g.T @ G_g_pre + self.W_o.T @ G_o_pre
```

```
            G_h = G_z[:self.hidden_dim]
            G_C = G_C_prev

        # Clip gradients to [-5, 5]
        for grad in [G_wf, G_bf, G_wi, G_bi, G_wg, G_bg, G_wo, G_bo, G_wy, G_by]:
            np.clip(grad, -5, 5, out=grad)

        # Update all weights
        ...

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

Why LSTM Works Better

1. **Cell state acts as a highway** - Gradients can flow through unimpeded
2. **Gates control information flow** - Network learns what to remember/forget
3. **Additive updates to cell state** - Reduces vanishing gradient problem

Training Example

```
lstm = LSTM(input_dim=1, hidden_dim=100, seq_len=250)

for epoch in range(num_epochs):
    start = np.random.randint(1, 701)
    inp = (np.arange(start, start + 250) / 1000).reshape(250, 1, 1)
    y_true = np.array([(start + 250) / 1000]).reshape(1, 1)

    y_pred = lstm.forward(inp)
    loss = 0.5 * (y_pred - y_true) ** 2

    lstm.backward(y_true, y_pred, lr=0.05)
```

The LSTM can handle much longer sequences (250 timesteps) compared to vanilla RNN (50 timesteps) due to its ability to maintain long-term dependencies.