A Complete Guide to LSTMs – From Why to How

Dhairya Kuchhal Har Ashish

June 20, 2025

Contents

1 Introduction **1**

2 What makes LSTM different? **2**
2.1 Forget me not! . . . 2
2.2 To Remember or not to Remember: That is the Question . . . 3

3 Architecture of LSTMs **3**

4 Forward Propogation **4**
4.1 Interpretation . . . 5
4.2 Implementation . . . 6

5 BackPropogation through time **8**
5.1 BPTT and Loss function . . . 8
5.2 Activation Function Derivatives . . . 8
5.3 Why do we go backward in time? . . . 9
5.4 Backpropagation Equations . . . 10
5.4.1 The cell state . . . 10
5.4.2 The gates . . . 10
5.4.3 Weights and biases . . . 12
5.4.4 The Hidden state and the cell state . . . 12
5.5 Implementation . . . 13
5.6 Prediction . . . 15

6 What have we achieved? **16**

1 Introduction

By now, you must have studied RNNs and how they are able to model sequential data using self-feedback loops and recurrence. However, standard RNNs cannot bridge more than 5–10 time steps - this is due to the fact that the back-propagated error signals tend to either grow or shrink with every time step, depending on the value of the error signal. Over many time steps, the error therefore typically blows up or vanishes. Blown-up error

signals lead straight to oscillating weights, whereas with a vanishing error, learning takes an unacceptable amount of time, or does not work at all.

Let's say an RNN computes hidden states like this:

$$\mathbf{h}_t = \tanh(\mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{W}_{xh}\mathbf{x}_t + \mathbf{b})$$

Here, allow us to provide a slighlty different approach to the vanishing/exploding gradients problem than that of whats you've already seen.

During training, you want to compute the gradient of the loss $\mathcal{L}$ with respect to earlier weights or inputs. That means you'll need to inevitably calculate the gradient with respect to $\mathbf{h}_k$ at some time $t = k$ when you utilise the chain rule. Let's calculate it directly here instead of going through the error signal δ_t^h:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{h}_k} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}_T} \cdot \prod_{t=k+1}^{T} \frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}}$$

This product of derivatives is the **problem**. Let's now zoom into the derivative:

$$\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-1}} \approx \mathbf{W}_{hh}^{\top} \cdot \text{diag}(1 - \mathbf{h}_{t-1}^2)$$

(This may look slightly different than you are used to because we are assuming the tanh activation function - best to get used to both sigmoid and tanh!)

- Here, `diag(1 - h²)` is the derivative of tanh (always ≤ 1).
- So you are **multiplying by something that shrinks** (≤ 1) **or explodes** (if eigenvalues of $W > 1$).

So imagine that you have 100 time steps. At each time step, you are multiplying the gradient with something less than 1, for example,, 0.8.

$$Gradient \sim (0.8)^{100} = 2.04 \times 10^{-10}$$

or with something greater than one, say 1.2 (which will eventually lead to loss becoming `NaN` and our training collapses).

$$Gradient \sim (1.2)^{100} \approx 8.2 \times 10^{7}$$

Hence, these flaws eventually led to the need of another model, which can model sequential data without such problems.

2 What makes LSTM different?

2.1 Forget me not!

To fix this problem of vanishing/exploding gradients, we need

- A memory element that preserves information over long time steps without degradation - that is, we can remember some information over longer periods of time before that information becomes useless through exploding/vanishing gradients.

- Selective update and read access so that the network knows *when* to remember, *when* to forget, and *when* to use to stored memory.

So firstly, in order to combat these problems, we need to find a way to let error flow back in time unchanged – i.e., a derivative of 1 over many time steps. Thus the need arose for a special neuron which just copies its value forward in time with weight 1.

$$C_{t+1} = C_t + \text{Something else } \textit{if necessary}$$

This became the constant error carousel (CEC)—a self-connected linear unit, whose value doesn't change *unless we choose to modify it.* We'll look more into this later.

2.2 To Remember or not to Remember: That is the Question

Now, imagine a naive memory cell that:

- Always stores whatever input it receives
- Always exposes its entire value to the rest of the network
- And is trained via backpropagation over long sequences

At first this would seem helpful, but it is accompanied by 2 major conflicts:

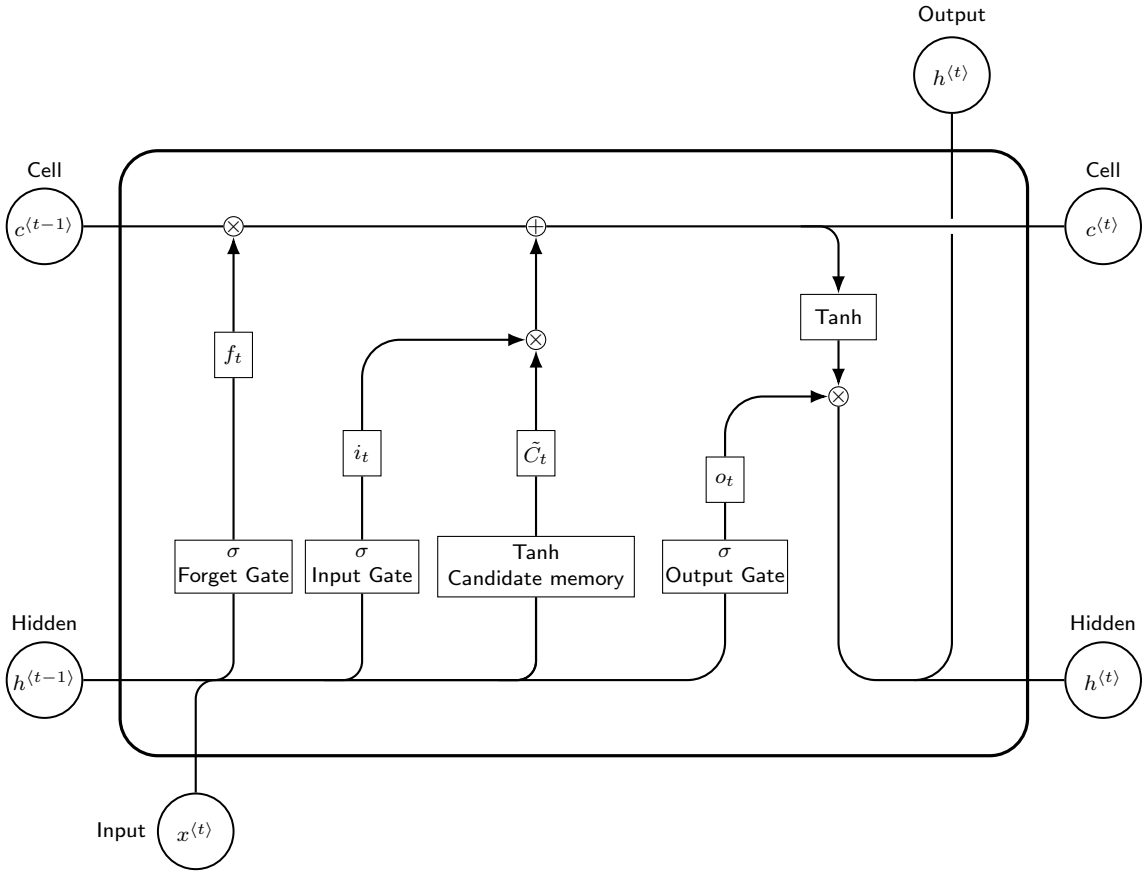
- ***Input Conflict:*** If the memory cell always accepts new inputs, then the same input weights must learn to **write new important information of memory**, as well as **not overwrite old information that needs to be preserved**. This is like trying to write in pen over something you may or may not want to erase without knowing whether the ink will dry or not. This leads to gradient intereference, instability and poor memory retention.
- ***Output Conflict:*** If the memory cell always outputs the full value, then the same output weights must learn to **read useful memory when it's needed** and **and suppress or hide irrelevant content at other times** hence we need a way to read from memory, as well as act as a noise gate to avoid leaking garbage into downstream calculations.

As you can understand, so just a all-remembering, all-telling neuron is not ideal for our work, hence LSTMs provide a mechanism to control **how much to remember, what we input, what we output and how much we output**. This mechanism is basically a combination of 3 gates called **forget gate, input gate and output gate.**

So basically, LSTM doesn't hardcode the importance of old memory, but instead learns patterns in data where remembering pays off and forgetting hurts (or vice versa) via gradient descent.

3 Architecture of LSTMs

Let's discuss the various components in LSTM model and what purpose does each one serve.



- ***Constant Error Carousel (CEC):*** This is the unbroken, self-connected linear memory line (on the top of the diagram) where gradient flow is preserved without decay (hence "constant error"). It is also called cell state, and represented by C_t.
- ***Forget Gate:*** This determines how much of the previous memory C_{t-1} should be retained. It takes in input as well as the previous hidden state, passes it through a squashing function (generally sigmoid), and multiplies it with the old cell state C_{t-1} in order to keep/erase the memory. It is represented by f_t.
- ***Candidate Memory:*** It is the potential memory which will be added to CEC after being modulated by the input gate. It is represented by $\tilde{C}_t$.
- ***Input Gate:*** This gate determines how much of the candidate memory should be added to the candidate memory should be added to CEC, in a manner similar to how forget gate works. This is represented by i_t.
- ***Output Gate:*** This gate manages how much/what portion of the updated memory should be visible as the output h_t (short term memory / hidden state). This is often represented by o_t.

4 Forward Propogation

Now, let's figure out how our inputs are propogated sequentially through time in LSTM

4.1 Interpretation

Let's begin with what we have already. Suppose, we are given h_{t-1} ie the hidden state from the previous time step, C_{t-1} the cell state of the previous time step and x_t the input of this time step. So let's proceed with how the math will work.

Firstly:

- $\mathbf{x}_t \in \mathbb{R}^d$: input vector at time t and d = dimension of input vector
- $\mathbf{h}_{t-1} \in \mathbb{R}^h$: hidden state from the previous time step
- $\mathbf{a}_t \in \mathbb{R}^h$: pre-activation input to the hidden layer
- $\mathbf{h}_t \in \mathbb{R}^h$: current hidden state (post-activation) / output
- σ : Sigmoid function
- $\mathbf{W}_k \in \mathbb{R}^{h \times d}$ and $k \in \{f, i, \tilde{c}, o\}$: are the weights which are multiplied to the input vector
- $\mathbf{U}_k \in \mathbb{R}^{h \times h}$ and $k \in \{f, i, \tilde{c}, o\}$: are the weights which are multiplied to the previous hidden state vector
- $\mathbf{b}_k \in \mathbb{R}^h$ and $k \in \{f, i, \tilde{c}, o\}$: are the biases added to pre-activation input

For the Forget Gate:

$$\begin{aligned} \mathbf{a}_{t_f} &= \mathbf{W}_f \mathbf{x}_t + \mathbf{U}_f \mathbf{h}_{t-1} + \mathbf{b}_f \\ \mathbf{f}_t &= \sigma(\mathbf{a}_{t_f}) \end{aligned}$$

For the Input Gate:

$$\begin{aligned} \mathbf{a}_{t_i} &= \mathbf{W}_i \mathbf{x}_t + \mathbf{U}_i \mathbf{h}_{t-1} + \mathbf{b}_i \\ \mathbf{i}_t &= \sigma(\mathbf{a}_{t_i}) \end{aligned}$$

For the Candidate Memory:

$$\begin{aligned} \mathbf{a}_{t_{\tilde{c}}} &= \mathbf{W}_{\tilde{c}} \mathbf{x}_t + \mathbf{U}_{\tilde{c}} \mathbf{h}_{t-1} + \mathbf{b}_{\tilde{c}} \\ \tilde{\boldsymbol{c}}_t &= Tanh(\mathbf{a}_{t_{\tilde{c}}}) \end{aligned}$$

For the Output Gate:

$$\begin{aligned} \mathbf{a}_{t_o} &= \mathbf{W}_o \mathbf{x}_t + \mathbf{U}_o \mathbf{h}_{t-1} + \mathbf{b}_o \\ \mathbf{o}_t &= \sigma(\mathbf{a}_{t_o}) \end{aligned}$$

For updating the cell state:

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\boldsymbol{c}}_{\boldsymbol{t}}$$

For updating the hidden state:

$$\mathbf{h}_t = \mathbf{o}_t \odot tanh(\mathbf{c}_t)$$

So, far, this is the mathematics of a single time step - computing the values of all the gates and then using them to update the cell and hidden state (Long and Short term memory). However, we still have to discuss how LSTMs are meant to propogate this data through time.

So now, we extend this forward pass to handle the entire sequence of input vectors $\{\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3....\mathbf{x}_T\}$ and compute the corresponding $\{\mathbf{h}_1, \mathbf{h}_2, \mathbf{h}_3....\mathbf{h}_T\}$. At each time step, the current hidden state ($\mathbf{h}_t$) and cell state($\mathbf{c}_t$) are updated which depend on the previous hidden state($\mathbf{h}_{t-1}$), previous cell state($\mathbf{c}_{t-1}$) and input ($\mathbf{x}_t$). We chain this across time and store all the intermediary values such as:

- Pre-activation states: $\{\mathbf{a}_{t_f}, \mathbf{a}_{t_i}, \mathbf{a}_{t_o}, \mathbf{a}_{t_{\tilde{c}}}\}$
- Gate Activations: $\{\mathbf{f}_t, \mathbf{i}_t, \mathbf{o}_t, \tilde{\boldsymbol{c}}_t\}$
- Hidden States: $\mathbf{h}_t$
- Cell States: $\mathbf{c}_t$

4.2 Implementation

The following code implements the initialisation of variables as well as full forward pass. X is the input sequence containing the whole sequence from time $t = 1$ to time$t = T$.

```
class LSTM:
    def __init__(self, d, h):
        self.input_dim = d
        self.hidden_dim = h
        scale = np.sqrt(2.0 / (d + h))

        self.W_f = np.random.randn(self.hidden_dim, self.input_dim) * scale
        self.U_f = np.random.randn(self.hidden_dim, self.hidden_dim) * scale
        self.b_f = np.random.randn(self.hidden_dim, 1) * scale

        self.W_i = np.random.randn(self.hidden_dim, self.input_dim) * scale
        self.U_i = np.random.randn(self.hidden_dim, self.hidden_dim) * scale
        self.b_i = np.random.randn(self.hidden_dim, 1) * scale

        self.W_can = np.random.randn(self.hidden_dim, self.input_dim) * scale
        self.U_can = np.random.randn(self.hidden_dim, self.hidden_dim) * scale
        self.b_can = np.random.randn(self.hidden_dim, 1) * scale

        self.W_o = np.random.randn(self.hidden_dim, self.input_dim) * scale
        self.U_o = np.random.randn(self.hidden_dim, self.hidden_dim) * scale
        self.b_o = np.random.randn(self.hidden_dim, 1) * scale

    def feedforward(self, X, y):
        self.time = X.shape[0]

        h_tminus1 = np.zeros((self.hidden_dim,1))
        c_tminus1 = np.zeros((self.hidden_dim,1))
```

```
28
29          self.h_all = [h_tminus1]
30          self.c_all = [c_tminus1]
31
32          self.f = [0]
33          self.i = [0]
34          self.o = [0]
35          self.can = [0]
36          self.a_f = [0]
37          self.a_i = [0]
38          self.a_o = [0]
39          self.a_can = [0]
40
41          for t in range(self.time):
42              X_t = X[t].reshape(self.input_dim, 1)
43
44              a_t_f = self.W_f @ X_t + self.U_f @ h_tminus1 + self.b_f
45              f_t = sigmoid(a_t_f)
46
47              a_t_i = self.W_i @ X_t + self.U_i @ h_tminus1 + self.b_i
48              i_t = sigmoid(a_t_i)
49
50              a_t_can = self.W_can @ X_t + self.U_can @ h_tminus1 + self.b_can
51              can_t = tanh(a_t_can)
52
53              a_t_o = self.W_o @ X_t + self.U_o @ h_tminus1 + self.b_o
54              o_t = sigmoid(a_t_o)
55
56              c_t = f_t * c_tminus1 + i_t * can_t
57              h_t = o_t * tanh(c_t)
58
59              c_tminus1 = c_t
60              h_tminus1 = h_t
61
62              self.c_all.append(c_tminus1)
63              self.h_all.append(h_tminus1)
64              self.f.append(f_t)
65              self.i.append(i_t)
66              self.o.append(o_t)
67              self.can.append(can_t)
68              self.a_f.append(a_t_f)
69              self.a_i.append(a_t_i)
70              self.a_o.append(a_t_o)
71              self.a_can.append(a_t_can)
72
73          return np.concatenate(self.h_all[1:], axis=1).T
```

A few things to explain more clearly what is going on. The first initialisation of the class, with the size of input as well as hidden states, leads to the initialisation of weights

and biases which will be used throughout the forward pass and training.

Then in the feedforward function, firstly we are reshaping the input vector for seamless matrix multiplication compatibility. We start with an initial hidden state ($\mathbf{h}_0$) and intial cell state ($\mathbf{c}_0$), which are typically initialised to a zero vector of shape (h, 1) where h is the hidden size. Then we implement the mathematics which we have discussed above and store all the values (such as Pre-activation states, gate activations, hidden states and Cell states) which we will need in the future for backpropagation through time. Lastly, our function returns a list which contains all the hidden states we have calculated (from time $t = 0$ to time $t = T$)

Hence, when we use our LSTM, we will initialise it once, to generate all the variables, and then for training we will call the feedforward function for all iteration.

5 BackPropogation through time

5.1 BPTT and Loss function

Training any ML algorithm involves minimising the loss function. The way to proceed about doing that is calculating the derviative of loss function with respect to all of out trainable parameters so as to do convex optimisation through gradient descent. This is **Backpropogation through time**

Similar to what we did in RNNs, we wish to minimse the total loss function. ie

$$\mathcal{L} = \sum_{t=1}^{T} \mathcal{L}_t$$

where $\mathcal{L}_t$ is the loss for the time step t, i.e.,

$$\mathcal{L}_t = \frac{1}{2} \|\hat{\mathbf{y}}_t - \mathbf{y}_t\|_2^2$$

Since the parameters are shared across all time steps, we **accumulate the gradients from each time step**. This means that for each parameter θ, the total gradient is:

$$\frac{\partial \mathcal{L}}{\partial \theta} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}_t}{\partial \theta}$$

We will not be showing this explicitly after every derivation, but it will be shown clearly in the code and explained along with it. The code for these is as follows:

```
1 def mse_loss(y_true, y_pred):
2   return ((y_true - y_pred) ** 2).mean()
```

5.2 Activation Function Derivatives

We will need these derivatives when we are performing the derivations for the back-propagation, so it is best to implement them now.

We had already seen in the forward pass that our activation function was the sigmoid activation function applied element-wise, that is:

$$\mathbf{h}_t = \sigma(\mathbf{a}_t) = \frac{1}{1 + e^{-\mathbf{a}_t}}$$

The derivative of the sigmoid activation function is:

$$\sigma'(\mathbf{a}_t) = \sigma(\mathbf{a}_t) \odot (1 - \sigma(\mathbf{a}_t)) = \mathbf{h}_t \odot (1 - \mathbf{h}_t)$$

and this itself is applied element-wise whenever we will use the chain rule, as in further up ahead.

If we were using tanh instead, its derivative would be:

$$\tanh'(\mathbf{a}_t) = 1 - \tanh^2(\mathbf{a}_t)$$

where 1 is accordingly broadcasted to the size of $\mathbf{a}_t$, inthis case, $\mathbb{R}^m$. The code for these is as follows:

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

def derive_tanh(x):
    return 1 - np.tanh(x)**2

def sigmoid(x):
    x = np.clip(x, -500, 500)
    return 1/(1+np.exp(-x))

def derive_sigmoid(x):
    f = sigmoid(x)
    return f*(1-f)
```

5.3 Why do we go backward in time?

One of the things which troubled me the most while coding LSTM (having just learnt the math) is that during BPTT why do we go from time $t = T$ to time $t = 1$ instead of $t = 1$ to $t = T$.

To answer this, let's see what all h_t depends on:

- Its own gate outputs at time t
- Its cell state $\mathbf{c}_t$
- The previous hidden state $\mathbf{h}_{t-1}$

so, when computing $\frac{\partial \mathcal{L}}{\partial \mathbf{h}_{t-1}}$ we need to know how it influenced $\mathbf{h}_t$, $\mathbf{c}_t$ and all future time steps. Thus, the chain rule forces you to go backwards to trace how every $\mathbf{h}_t$ or $\mathbf{c}_t$ affects the final loss.

Now, let's give it a mathematical analogy of chain rule of differentiation. Let's say $\mathcal{L}$ depends on $\mathbf{h}_3$, $\mathbf{h}_3$ depends on $\mathbf{h}_2$, $\mathbf{h}_2$ depends on $\mathbf{h}_1$. Hence to calculate $\frac{\partial \mathcal{L}}{\partial \mathbf{h}_1}$, we need to calculate $\frac{\partial \mathcal{L}}{\partial \mathbf{h}_3}, \frac{\partial \mathbf{h}_3}{\partial \mathbf{h}_2}, \frac{\partial \mathbf{h}_2}{\partial \mathbf{h}_1}$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{h}_1} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}_3} . \frac{\partial \mathbf{h}_3}{\partial \mathbf{h}_2} . \frac{\partial \mathbf{h}_2}{\partial \mathbf{h}_1}$$

but we cannot go around calculating $\frac{\partial \mathcal{L}}{\partial \mathbf{h}_3}$ by calculating $\frac{\partial \mathcal{L}}{\partial \mathbf{h}_1}, \frac{\partial \mathbf{h}_1}{\partial \mathbf{h}_2}, \frac{\partial \mathbf{h}_2}{\partial \mathbf{h}_3}$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{h}_3} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}_1} . \frac{\partial \mathbf{h}_1}{\partial \mathbf{h}_2} . \frac{\partial \mathbf{h}_2}{\partial \mathbf{h}_3}$$

5.4 Backpropagation Equations

5.4.1 The cell state

Herein, we assume that we are given $\frac{\partial \mathcal{L}}{\partial \mathbf{h}_t}$ and $\frac{\partial \mathcal{L}}{\partial \mathbf{c}_{t+1}}$. The derivative of the loss $\mathcal{L}$ with respect to the cell state $\mathbf{c}_t$ (denoted $\boldsymbol{\delta}_{\mathbf{c}_t}$) receives two contributions:

- From the **current time step**, because c_t affects the hidden state h_t:

  $$\mathbf{h}_t = \mathbf{o}_t \odot tanh(\mathbf{c}_t)$$

  Taking derivative with respect to $\mathbf{c}_t$

  $$\frac{\partial \mathbf{h}_t}{\partial \mathbf{c}_t} = \mathbf{o}_t \odot \frac{\partial (Tanh(\mathbf{c}_t))}{\partial c_t} = \mathbf{o}_t \odot (1 - tanh^2(\mathbf{c}_t))$$

  hence, $\frac{\partial \mathcal{L}}{\partial c_t}$ comes out to be

  $$\left.\frac{\partial \mathcal{L}}{\partial c_t}\right|_{\text{via } \mathbf{h}_t} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}_t} \cdot \frac{\partial \mathbf{h}_t}{\partial \mathbf{c}_t} = \frac{\partial \mathcal{L}}{\partial \mathbf{h}_t} \odot \mathbf{o}_t \odot (1 - tanh^2(\mathbf{c}_t))$$

- From the **future time step**, because C_t directly contributes to C_{t+1} via the forget gate f_{t+1} from the equation:

  $$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\boldsymbol{c}}_{\boldsymbol{t}}$$

  Taking derivative with respect to $\mathbf{c}_t$ gives us:

  $$\frac{\partial \mathbf{c}_{t+1}}{\partial \mathbf{c}_t} = \mathbf{f}_{t+1}$$

  Hence, $\frac{\partial \mathcal{L}}{\partial c_t}$ comes out to be:

  $$\left.\frac{\partial \mathcal{L}}{\partial \mathbf{c}_t}\right|_{\text{via } \mathbf{c}_{t+1}} = \frac{\partial \mathcal{L}}{\partial \mathbf{c}_{t+1}} \cdot \frac{\partial \mathbf{c}_{t+1}}{\partial \mathbf{c}_t} = \boldsymbol{\delta}_{\mathbf{c}_{t+1}} \odot \mathbf{f}_{t+1}$$

Therefore, the total gradient $\frac{\partial \mathcal{L}}{\partial \mathbf{c}_t}$ becomes:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{c}_t} = \underbrace{\frac{\partial \mathcal{L}}{\partial \mathbf{h}_t} \odot \mathbf{o}_t \odot (1 - tanh^2(\mathbf{c}_t))}_{\text{from current } h_t} + \underbrace{\boldsymbol{\delta}_{\mathbf{c}_t}}_{\text{already accumulated from future}}$$

5.4.2 The gates

Keeping these equation in mind

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\boldsymbol{c}}_{\boldsymbol{t}}$$

$$\mathbf{h}_t = \mathbf{o}_t \odot tanh(\mathbf{c}_t)$$

We take the derivative, of $\mathbf{c}_t$ with respect to $\mathbf{f}_t$, $\mathbf{i}_t$ and $\tilde{\boldsymbol{c}}_t$, as well as the derivative of $\mathbf{h}_t$ with respect to $\mathbf{o}_t$:

$$\frac{\partial \boldsymbol{c}_t}{\partial \boldsymbol{f}_t} = \boldsymbol{c}_{t-1}$$
$$\frac{\partial \boldsymbol{c}_t}{\partial \boldsymbol{i}_t} = \tilde{\boldsymbol{c}}_t$$
$$\frac{\partial \boldsymbol{c}_t}{\partial \tilde{\boldsymbol{c}}_t} = \boldsymbol{i}_t$$
$$\frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{o}_t} = tanh(\mathbf{c}_t)$$

Hence, finally taking derivative of loss with respect to gate activations:

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial \boldsymbol{f}_t} &= \frac{\partial \mathcal{L}}{\partial \boldsymbol{c}_t} \odot \frac{\partial \boldsymbol{c}_t}{\partial \boldsymbol{f}_t} \\ &= \frac{\partial \mathcal{L}}{\partial \boldsymbol{c}_t} \odot \boldsymbol{c}_{t-1}\end{aligned}$$

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{i}_t} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{c}_t} \odot \frac{\partial \boldsymbol{c}_t}{\partial \boldsymbol{i}_t} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{c}_t} \odot \tilde{\boldsymbol{c}}_t$$

$$\frac{\partial \mathcal{L}}{\partial \tilde{\boldsymbol{c}}_t} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{c}_t} \odot \frac{\partial \boldsymbol{c}_t}{\partial \tilde{\boldsymbol{c}}_t} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{c}_t} \odot \boldsymbol{i}_t$$

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{o}_t} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{h}_t} \odot \frac{\partial \boldsymbol{h}_t}{\partial \boldsymbol{o}_t} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{h}_t} \odot tanh(\mathbf{c}_t)$$

Taking derivative of gate activation with respect to their preactivation function:

$$\frac{\partial \boldsymbol{f}_t}{\partial \boldsymbol{a}_{t_f}} = \boldsymbol{f}_t \odot (1 - \boldsymbol{f}_t)$$

$$\frac{\partial \boldsymbol{i}_t}{\partial \boldsymbol{a}_{t_i}} = \boldsymbol{i}_t \odot (1 - \boldsymbol{i}_t)$$

$$\frac{\partial \tilde{\boldsymbol{c}}_t}{\partial \boldsymbol{a}_{t_{\tilde{c}}}} = (1 - \tilde{\boldsymbol{c}}_t^2)$$

$$\frac{\partial \boldsymbol{o}_t}{\partial \boldsymbol{a}_{t_o}} = \boldsymbol{o}_t \odot (1 - \boldsymbol{o}_t)$$

Thus, error in the pre activation states of gates are:

$$\boldsymbol{\delta}_{\boldsymbol{f_t}} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{f}_t} . \frac{\partial \boldsymbol{f}_t}{\partial \boldsymbol{a}_{t_f}} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{f}_t} \odot \boldsymbol{f}_t \odot (1 - \boldsymbol{f}_t)$$

$$\boldsymbol{\delta}_{\boldsymbol{i_t}} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{i}_t} . \frac{\partial \boldsymbol{i}_t}{\partial \boldsymbol{a}_{t_i}} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{i}_t} \odot \boldsymbol{i}_t \odot (1 - \boldsymbol{i}_t)$$

$$\boldsymbol{\delta}_{\tilde{\boldsymbol{c}}_t} = \frac{\partial \mathcal{L}}{\partial \tilde{\boldsymbol{c}}_t} . \frac{\partial \tilde{\boldsymbol{c}}_t}{\partial \boldsymbol{a}_{t_{\tilde{c}}}} = \frac{\partial \mathcal{L}}{\partial \tilde{\boldsymbol{c}}_t} \odot (1 - \tilde{\boldsymbol{c}}_t^2)$$

$$\boldsymbol{\delta}_{\boldsymbol{o_t}} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{o}_t} . \frac{\partial \boldsymbol{o}_t}{\partial \boldsymbol{a}_{t_o}} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{o}_t} \odot \boldsymbol{o}_t \odot (1 - \boldsymbol{o}_t)$$

Sanity Check: As you might have seen, in this sub-sub-section we have always used element wise multiplication, from which we can say that all shape of all these vectors hasn't changed one bit, and all of these vectors $\in \mathbb{R}^h$

5.4.3 Weights and biases

Now, finally, we take the derivative of the pre-activation states of gates with respect to weights and biases:

$$\frac{\partial \boldsymbol{a}_{t_k}}{\partial \boldsymbol{W}_k} = x_t^T$$
$$\frac{\partial \boldsymbol{a}_{t_k}}{\partial \boldsymbol{U}_k} = h_{t-1}^T$$
$$\frac{\partial \boldsymbol{a}_{t_k}}{\partial \boldsymbol{b}_k} = 1$$

where $k \in \{\boldsymbol{f}, \boldsymbol{i}, \boldsymbol{o}, \tilde{\boldsymbol{c}}\}$

Then, we calculate the derivative of loss with respect to these weights and biases:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{W}_k} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{a}_{t_k}} . \frac{\partial \boldsymbol{a}_{t_k}}{\partial \boldsymbol{W}_k} = \boldsymbol{\delta}_k . \boldsymbol{x}_t^T$$

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{U}_k} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{a}_{t_k}} . \frac{\partial \boldsymbol{a}_{t_k}}{\partial \boldsymbol{U}_k} = \boldsymbol{\delta}_k . \boldsymbol{h}_{t-1}^T$$

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{b}_k} = \frac{\partial \mathcal{L}}{\partial \boldsymbol{a}_{t_k}} . \frac{\partial \boldsymbol{a}_{t_k}}{\partial \boldsymbol{b}_k} = \boldsymbol{\delta}_k$$

where $k \in \{\boldsymbol{f}, \boldsymbol{i}, \boldsymbol{o}, \tilde{\boldsymbol{c}}\}$

5.4.4 The Hidden state and the cell state

Now, lastly, we calculate the derivative of loss with respect to hidden state and cell state of previous time step:

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{h}_{t-1}} = \boldsymbol{U}_f^T . \boldsymbol{\delta_f} + \boldsymbol{U}_i^T . \boldsymbol{\delta_i} + \boldsymbol{U}_o^T . \boldsymbol{\delta_o} + \boldsymbol{U}_{\tilde{c}}^T . \boldsymbol{\delta}_{\tilde{\boldsymbol{c}}}$$

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{c}_{t-1}} = \boldsymbol{W}_f^T . \boldsymbol{\delta_f} + \boldsymbol{W}_i^T . \boldsymbol{\delta_i} + \boldsymbol{W}_o^T . \boldsymbol{\delta_o} + \boldsymbol{W}_{\tilde{c}}^T . \boldsymbol{\delta}_{\tilde{\boldsymbol{c}}}$$

5.5 Implementation

Now that we have understood all the matrix calculus of gradient descent we use in LSTM and calculated the gradients, let's figure out how we will implement these in our model. These above calculations just show the gradient descent for 1 time step, but in 1 iteration, we will need to do so for all time steps. Hence, we will start calculating the derivative of loss with respect to each of the parameters and start summing those up in order to get the cumulative gradients and finally subtract those from the respective parameters. The code for this is given below:

```
def train(self, X, y, n_epochs = 100, lr = 0.01):
        T = X.shape[0]
        N = X.shape[1]
        X = X.reshape(T, N, 1)
        y = y.reshape(T, 1)

        X_mean, X_std = X.mean(), X.std()
        y_mean, y_std = y.mean(), y.std()
        X = (X - X_mean) / (X_std + 1e-8)
        y = (y - y_mean) / (y_std + 1e-8)


        for epoch in range(n_epochs):
            d_W_f = np.zeros_like(self.W_f)
            d_W_i = np.zeros_like(self.W_i)
            d_W_o = np.zeros_like(self.W_o)
            d_W_can = np.zeros_like(self.W_can)

            d_U_f = np.zeros_like(self.U_f)
            d_U_i = np.zeros_like(self.U_i)
            d_U_o = np.zeros_like(self.U_o)
            d_U_can = np.zeros_like(self.U_can)

            d_b_f = np.zeros_like(self.b_f)
            d_b_i = np.zeros_like(self.b_i)
            d_b_o = np.zeros_like(self.b_o)
            d_b_can = np.zeros_like(self.b_can)
            preds = self.feedforward(X, y)

            total_loss = mse_loss(y, preds)

            dh_next = np.zeros((self.hidden_dim, 1))
            dC_next = np.zeros((self.hidden_dim, 1))

            for t in reversed(range(1, T+1)):

                x_t = X[t-1]

                dh_t = (self.h_all[t] - y[t - 1]) + dh_next
```

```
40
41                  dC_t = dh_t * self.o[t] * derive_tanh(self.c_all[t]) + dC_next
42
43                  d_f = dC_t * self.c_all[t-1]
44                  d_i = dC_t * self.can[t]
45                  d_can = dC_t * self.i[t]
46
47                  delta_f = d_f*derive_sigmoid(self.a_f[t])
48                  delta_i = d_i*derive_sigmoid(self.a_i[t])
49                  delta_o = dh_t*tanh(self.c_all[t])* derive_sigmoid(self.a_o[t])
50                  delta_can = d_can*derive_tanh(self.a_can[t])
51
52                  d_W_f += delta_f @ x_t.T
53                  d_W_i += delta_i @ x_t.T
54                  d_W_o += delta_o @ x_t.T
55                  d_W_can += delta_can @ x_t.T
56
57                  d_U_f += delta_f @ self.h_all[t-1].T
58                  d_U_i += delta_i @ self.h_all[t-1].T
59                  d_U_o += delta_o @ self.h_all[t-1].T
60                  d_U_can += delta_can @ self.h_all[t-1].T
61
62                  d_b_f += delta_f
63                  d_b_i += delta_i
64                  d_b_o += delta_o
65                  d_b_can += delta_can
66
67                  dh_t = self.U_f.T @ delta_f + self.U_i.T @ delta_i + self.U_o.T @ del
68
69                  dC_next = dC_t * self.f[t]
70              grad_norm = np.sqrt(np.sum(d_W_f**2) + np.sum(d_W_i**2) + np.sum(d_W_o**2
71                                  np.sum(d_U_f**2) + np.sum(d_U_i**2) + np.sum(d_U_o**2)
72              if grad_norm > 5.0:
73                  clip_factor = 5.0 / grad_norm
74                  d_W_f *= clip_factor
75                  d_W_i *= clip_factor
76                  d_W_o *= clip_factor
77                  d_W_can *= clip_factor
78                  d_U_f *= clip_factor
79                  d_U_i *= clip_factor
80                  d_U_o *= clip_factor
81                  d_U_can *= clip_factor
82                  d_b_f *= clip_factor
83                  d_b_i *= clip_factor
84                  d_b_o *= clip_factor
85                  d_b_can *= clip_factor
86              self.W_f -= lr*d_W_f
87              self.W_i -= lr*d_W_i
```

```
            self.W_o -= lr*d_W_o
            self.W_can -= lr*d_W_can

            self.U_f -= lr*d_U_f
            self.U_i -= lr*d_U_i
            self.U_o -= lr*d_U_o
            self.U_can -= lr*d_U_can

            self.b_f -= lr*d_b_f
            self.b_i -= lr*d_b_i
            self.b_o -= lr*d_b_o
            self.b_can -= lr*d_b_can
            if epoch%10 == 9 or epoch == n_epochs or epoch == 0:
                print(f"Epoch {epoch + 1}, Loss: {total_loss:.4f}")
```

So let's discuss what we are doing here.

- Firstly, we reshape and scale the data, so as to make the training more efficient and prevent integer overflow.
- Then, in the loop of iterations, we initialise the matrices/vectors in which the gradient for each timestep will be added.
- Then, we initialise the $\frac{\partial \mathcal{L}}{\partial \mathbf{h}_t}$ and $\frac{\partial \mathcal{L}}{\partial \mathbf{c}_{t+1}}$ vectors with 0 value so as to maintain the loop.
- A loop is initialised from time T to time 1, in which all the gradients are calculated and then added to the summing matrices.
- After going through this time loop, we calculate the l-2 norm of these gradients, and then proceed to clip these gradients if they are too much, so as to prevent integer overflow and destabilisation of the learning algorithm
- Lastly, we subtract these gradients from our parameters and then proceed with the next iteration.

5.6 Prediction

When trying to predict, we just need to feed forward the inputs we have in out prediction sequence, and we will get the output as the hidden states of the sequence. This should be fairly easy as we already know the forward pass mathematics. Here is the code for it:

```
def predict(self, X):
        n_samples = X.shape[0]
        sequence_length = X.shape[1]
        X = X.reshape(n_samples, sequence_length, 1)
        predictions = []

        h_t = np.zeros((self.hidden_dim, 1))
        c_t = np.zeros((self.hidden_dim, 1))

        for t in range(n_samples):
```

```
            x_t = X[t]
            f_t = sigmoid(self.W_f @ x_t + self.U_f @ h_t + self.b_f)
            i_t = sigmoid(self.W_i @ x_t + self.U_i @ h_t + self.b_i)
            can_t = tanh(self.W_can @ x_t + self.U_can @ h_t + self.b_can)
            o_t = sigmoid(self.W_o @ x_t + self.U_o @ h_t + self.b_o)
            c_t = f_t * c_t + i_t * can_t
            h_t = o_t * tanh(c_t)
            predictions.append(h_t)
        return np.array(predictions)

```

6 What have we achieved?

Congratulations, If you have understood and implemented everything perfectly up till now, you now have your LSTM ready for training. So, let's do that now.

Let's try to model the sinusoidal sequence which our RNN failed to capture in the previous paper. In this training data, we are given the value of sin(t) and we have to predict the value of sin(t+1) which is something fairly easy for even an linear model as $sin(t+1) = sin(t)cos(1) + cos(t)sin(1)$ but let's try it here:

To generate the training and testing data:

```
def generate_sine_sequence(seq_length, start = 0,  step_size=0.1):
    t = np.arange(start, seq_length) * step_size
    X = np.sin(t[:-1])
    Y = np.sin(t[1:])
    X = X.reshape(len(X), 1,1)
    return X, Y

# Create sine data
X, Y = generate_sine_sequence(seq_length=1000)
test_X, test_y = generate_sine_sequence(start=1000, seq_length=1300)
```

Now, let's initialise the LSTM and train it on our training data:

```
lstm = LSTM(1,1)
lstm.train(X, Y, lr = 0.01, n_epochs = 130)
```

What's the output that we get?

```
Epoch 1, Loss: 0.6425
Epoch 10, Loss: 0.4398
Epoch 20, Loss: 0.2590
Epoch 30, Loss: 0.1752
Epoch 40, Loss: 0.1290
Epoch 50, Loss: 0.0998
Epoch 60, Loss: 0.0819
Epoch 70, Loss: 0.0711
Epoch 80, Loss: 0.0643
Epoch 90, Loss: 0.0609
```

```
Epoch 100, Loss: 0.0644
Epoch 110, Loss: 0.0678
Epoch 120, Loss: 0.0707
Epoch 130, Loss: 0.0731
```

Now that we have trained our LSTM, let's try to predict something using our model:

```
from sklearn.metrics import r2_score
preds = lstm.predict(test_X)
preds = preds.reshape(299,)
print("MSE Loss: ", mse_loss(preds, test_y))
print("R^2 Score: ",r2_score(preds, test_y))
```

We get the output:

```
MSE Loss:  0.010389265962532031
R^2 Score:  0.9747084632006038
```

which is quite good.

To know what we are dealing with here, let's try to plot the y values (sin(t+1)) with the t+1 values and see what we get.

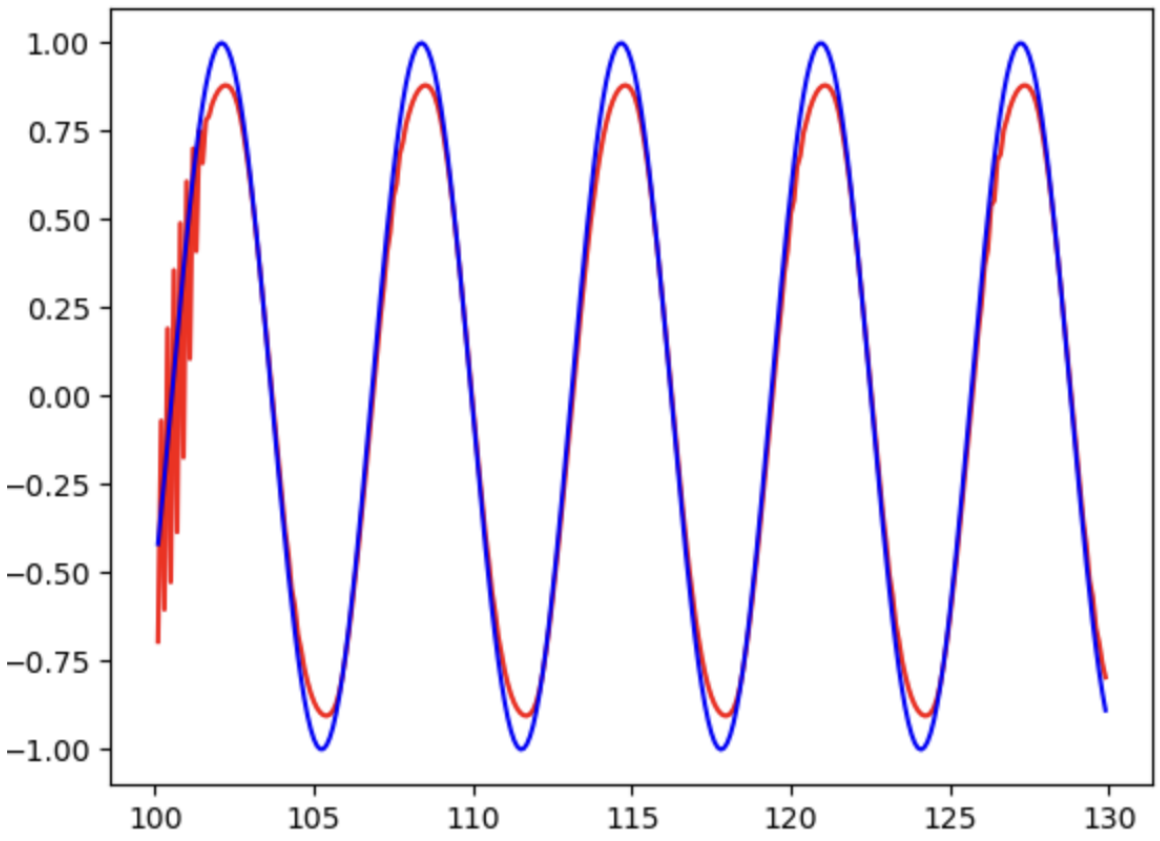


Figure 1: Blue = Actual Sin values, Red = Our predictions

As you can see, what we have achieved is very promising and gives us a glimpse into what LSTMs can do.