Introduction to LSTM

The RNN Problem

🔷 Gradient Flow in RNNs

- In **backpropagation through time (BPTT)**, the gradient has to pass backward through **many time steps**.
- Each step multiplies the gradient by weight matrices (e.g., W_{hh}) and derivatives of activations (like tanh).
- This repeated multiplication causes:

1. **Vanishing Gradient**
    - Gradients shrink → earlier layers (long-term past) get almost no updates.
    - RNNs fail to learn **long-term dependencies**.
2. **Exploding Gradient**
    - Gradients grow exponentially → unstable training.
    - Often fixed by **gradient clipping**.

Short-Term vs Long-Term Dependencies

- **Short-term dependency**: The output depends on inputs from the **recent past**.

  Example:

    - Sentence: *"The cat sat on the ___"*.
    - Predicting the next word "mat" mainly depends on the last 2–3 words.
- **Long-term dependency**: The output depends on inputs from the **distant past**.

  Example:

    - Sentence: *"I grew up in France … I speak fluent ___"*.

- To predict “French”, the model must remember “France” from far back.
- **Vanilla RNNs usually fail here** because the gradient fades over many steps.

Enter LSTM (Long Short-Term Memory)

LSTM is a special type of RNN architecture designed to overcome the **vanishing gradient problem** and handle long-term dependencies effectively.

📥 Core Idea

- Instead of a single hidden state, LSTMs maintain a **cell state** (C_t) that acts like a “conveyor belt” carrying long-term memory through the sequence with minimal changes.
- Gates control what information is added, removed, or output from memory.

LSTM Architecture

Quick overview / purpose

An **LSTM (Long Short-Term Memory)** cell is a recurrent unit that maintains two things each time step:

- a **cell state** C_t (long-term memory / “conveyor belt”), and
- a **hidden state** h_t (short-term memory / what’s “exposed” to the outside).

LSTM uses **gates** (sigmoid units) to control what gets written, kept, or read from the cell state — this is what lets it learn long-range dependencies better than a vanilla RNN.

Notation & dimensions

- Input at time t: $x_t \in \mathbb{R}^D$
- Hidden size: H

- Hidden state: $h_t \in \mathbb{R}^H$
- Cell state: $C_t \in \mathbb{R}^H$
- You'll see weight matrices for each gate. Two common parameterizations:
  - **Separate matrices per gate** (clearer): e.g. $W_f \in \mathbb{R}^{H\times D},\ U_f \in \mathbb{R}^{H\times H},\ b_f \in \mathbb{R}^H$.
  - **Combined matrices** (efficient): stack the 4 gates into one matmul: $W \in \mathbb{R}^{4H\times D},\ U \in \mathbb{R}^{4H\times H},\ b \in \mathbb{R}^{4H}$.

Canonical LSTM equations (separate matrices)

At each time step:

1. **Forget gate —** what to forget from previous cell:

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

2. **Input (update) gate —** how much new information to write:

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

3. **Candidate (new content) —** new information proposed for the cell:

$$\tilde{C}_t = \tanh(W_c x_t + U_c h_{t-1} + b_c)$$

4. **Cell state update —** combine old and new:

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

(Here $\odot$ is element-wise product.)

5. **Output gate —** decide what to output:

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

6. **Hidden state (the exposed output of the cell)**:

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

If you compute all four in one go (stacked), you do two big matmuls: W x_t and U h_{t-1}, add b, then split into four chunks and apply activations.

Component-by-component detail & intuition

Forget gate f_t

- Range: elements in (0,1) by sigmoid.
- Purpose: **scale** the previous cell C_{t-1}. If an element of f_t is near 0, that dimension of C_{t-1} is erased; if near 1, it's retained.
- Intuition: selective forgetting — the network learns *when not to keep old info*.

Input gate i_t and candidate $\tilde{C}_t$

- i_t (sigmoid) decides **how much of the candidate** to write into the cell.
- $\tilde{C}_t$ (tanh) is the new candidate content (in [-1,1]).
- Together, $i_t \odot \tilde{C}_t$ is the update term added into the cell.

Cell state C_t (the conveyor belt)

- Crucial property: **linear path** with elementwise gating,

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

  which allows information (and gradients) to flow across many time steps with limited distortion.
- Because the update is a *sum* of retained memory and new content, gradients can flow back through the addition instead of repeated nonlinear compressions.

Output gate o_t and hidden state h_t

- o_t decides which parts of the (squashed) cell state to expose as hidden output.
- $h_t = o_t \odot \tanh(C_t)$ — this is typically fed to the next layer (or used to predict y_t).

Why LSTM helps vanishing gradients (math intuition)

When backpropagating, the derivative of C_t w.r.t. C_{t-1} is:

$$\frac{\partial C_t}{\partial C_{t-1}} = \mathrm{diag}(f_t)$$

So the signal is multiplied elementwise by f_t (not by recurrent weight matrices and repeated tanh derivatives). If the network learns $f_t \approx 1$ for relevant dimensions, gradients are preserved across many steps — this is the **constant error carousel** idea.

Variants & extensions (brief)

- **Peephole connections:** gates also receive the previous cell C_{t-1} (terms like $V_f \odot C_{t-1}$), giving gates direct access to cell content.
- **Coupled input-forget gates:** sometimes $i_t = 1 - f_t$ to reduce parameters.
- **GRU (Gated Recurrent Unit):** simpler 2-gate variant (update + reset) with one hidden state (no separate cell state).
- **Bidirectional LSTM:** process sequence forwards and backwards and concatenate outputs.
- **Stacked LSTM:** multiple LSTM layers stacked for hierarchical features.
- **Regularization:** dropout between layers, recurrent dropout variants, layer normalization.

Implementation & performance tips

- **Efficient compute:** combine gates into single matmuls for Wx_t and Uh_{t-1}, then split — this is how frameworks implement it.
- **Initialization:** biases for forget gate b_f are often initialized to a positive value (e.g., 1) to encourage remembering at start of training.
- **Training:** still use gradient clipping to handle exploding gradients; use truncated BPTT (limit sequence length) for long sequences.
- **Batching / padded sequences:** handle variable lengths with masks or packed sequences.

Short pseudocode (forward, high level)

```
for t in 1..T:
    z = W @ x_t + U @ h_{t-1} + b      # z shape = 4H
    f, i, o, g = split(z)              # apply σ, σ, σ, tanh respectively
    C_t = f * C_{t-1} + i * g
    h_t = o * tanh(C_t)
```

Summary (one paragraph)

An LSTM augments the vanilla RNN with a **cell state** and **learned gates** (forget, input, output) that control reading, writing and erasing memory. The gates are elementwise sigmoids and the candidate uses tanh. Because the cell state update uses gated *additions* rather than repeated nonlinear transforms, gradients can flow much more freely across long sequences — solving (or greatly reducing) the vanishing gradient problem and enabling learning of long-term dependencies.

Why LSTM is Better than Vanilla RNN

Problem in Vanilla RNN	How LSTM Fixes It
Vanishing Gradient	The cell state has **linear paths** with controlled gates → allows gradients to flow back without vanishing quickly.

Problem in Vanilla RNN	How LSTM Fixes It
Forgetting old info	Forget gate explicitly decides what to erase.
Storing long-term info	Input + cell state keep useful info across long sequences.
Short-term vs Long-term	Gates balance between remembering recent info and far-past info.

Summary with Intuition

- **Vanilla RNN**: Think of it like writing on paper with an eraser that smudges over time — the old info fades.
- **LSTM**: Think of it like a notebook with tabs (gates):
    - *Forget gate*: erase old notes.
    - *Input gate*: add new notes.
    - *Output gate*: decide which notes to share now.
- This makes LSTM capable of learning **both short-term and long-term dependencies**.

LSTM Training

1. Forward pass (recap)

At each timestep tt, an LSTM computes:

$$
\begin{aligned}
f_t &= \sigma(W_f x_t + U_f h_{t-1} + b_f) \\
i_t &= \sigma(W_i x_t + U_i h_{t-1} + b_i) \\
\tilde{C}_t &= \tanh(W_c x_t + U_c h_{t-1} + b_c) \\
C_t &= f_t \odot C_{t-1} + i_t \odot \tilde{C}_t \\
o_t &= \sigma(W_o x_t + U_o h_{t-1} + b_o) \\
h_t &= o_t \odot \tanh(C_t) \\
y_t &= W_{hy} h_t + b_y
\end{aligned}
$$

---

2. Loss calculation

Suppose you have a sequence of length T, with predictions $\{y_t\}$ and targets $\{\hat{y}_t\}$.

The total loss is usually the **sum (or mean)** over timesteps:

$$\mathcal{L} = \sum_{t=1}^{T} \ell(y_t, \hat{y}_t)$$

---

3. Backpropagation Through Time (BPTT) — general idea

- You **unroll the LSTM across timesteps** (like a deep feedforward network with shared weights).
- Then you apply standard backpropagation *through the unrolled graph*.
- The challenge: **gradients at timestep tt depend not just on hth_t, but also indirectly on all previous hidden and cell states**.

So, the error at time tt flows back through:

- The output layer weights (W_{hy})
- The output gate and cell state at time t
- The **previous hidden state** h_{t-1} and **previous cell state** C_{t-1}
- Repeated until you reach the start of the sequence.

---

4. Gradients in LSTM (math intuition)

Key idea: two "paths" for error flow

- Via **hidden state** h_t (nonlinear, can still vanish/explode).
- Via **cell state** C_t (linear + forget gate, reduces vanishing gradient).

---

1. Gradient flow for cell state

   From the update:

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

   The derivative is:

   $$\frac{\partial C_t}{\partial C_{t-1}} = f_t$$

   So, when you backpropagate:

   $$\frac{\partial \mathcal{L}}{\partial C_{t-1}} = \frac{\partial \mathcal{L}}{\partial C_t} \odot f_t$$

   👉 This means **if $f_t \approx 1$**, the gradient flows backward almost unchanged — solving the vanishing gradient problem.

---

2. Gradients for gates

   Each gate gets its own error signal:

   - Forget gate:

   $$\frac{\partial \mathcal{L}}{\partial f_t} = \frac{\partial \mathcal{L}}{\partial C_t} \odot C_{t-1}$$

   - Input gate:

   $$\frac{\partial \mathcal{L}}{\partial i_t} = \frac{\partial \mathcal{L}}{\partial C_t} \odot \tilde{C}_t$$

- Candidate:

$$\frac{\partial \mathcal{L}}{\partial \tilde{C}_t} = \frac{\partial \mathcal{L}}{\partial C_t} \odot i_t$$

- Output gate:

$$\frac{\partial \mathcal{L}}{\partial o_t} = \frac{\partial \mathcal{L}}{\partial h_t} \odot \tanh(C_t)$$

Then you backprop through the activations (sigmoid/tanh), and compute parameter updates using:

$$\Delta W = \sum_t \frac{\partial \mathcal{L}}{\partial W}$$

---

5. Tricks used in practice

- **Truncated BPTT:** Instead of unrolling for the entire sequence, cut it into smaller chunks (e.g. 20–50 steps). Prevents memory blowup and helps stability.
- **Gradient clipping:** Prevents exploding gradients by capping the gradient norm.
- **Bias init for forget gate:** Often set $b_f \approx 1$ so that ftf_t starts near 1 (helps preserve memory early in training).
- **Layer normalization/dropout:** Help stabilize training.

✅ Summary

Training LSTMs = run forward pass (compute hidden + cell states), compute loss, and apply **BPTT** by unrolling across timesteps.

- Gradients flow both through the hidden states and the cell state.
- The **cell state provides a nearly-linear highway for gradients**, controlled by forget gates — this is why LSTMs handle long-term dependencies better than vanilla RNNs.

- Practical training uses truncated BPTT + gradient clipping for efficiency and stability.

Main Drawbacks of LSTMs

🧩 A. Architectural Limitations

1️⃣ *Sequential Processing (No Parallelization)*

- LSTMs process sequences **token by token**, where each step depends on the previous one.
- This means you **can't parallelize** time steps — only batches.
- So, training is **slow** compared to models like Transformers that process entire sequences simultaneously via self-attention.

> 💬 Example:
>
> If your sentence has 200 words, the LSTM must process them one by one — unlike Transformers that process all 200 words in parallel.

2️⃣ *Difficulty with Very Long Dependencies*

- Even though LSTMs improve over vanilla RNNs, they **still struggle** with dependencies spanning hundreds or thousands of time steps.
- The memory cell helps, but it's not perfect — gradients can still diminish over very long sequences.

> 💬 Example:
>
> In a document, connecting a noun introduced 100 sentences earlier to a pronoun now is often beyond an LSTM's capacity.

3️⃣ *Fixed-Length Hidden State Bottleneck*

- All sequence information must be compressed into a **fixed-size hidden vector** (`h_t`).

- This forces the model to “stuff” all context into a limited-size memory — leading to **information loss** for long texts.

⚙️ B. Computational Limitations

4️⃣ *Slow Training and Inference*

- Sequential nature + many gates → computationally heavy.
- Backpropagation through time (BPTT) makes gradient updates slow.
- More parameters per cell than simple RNNs or GRUs.

5️⃣ *Memory Usage*

- LSTMs store multiple gates (input, forget, output, and cell) per timestep, so memory consumption can be high.
- Especially problematic for long sequences or large batch sizes.

🧠 C. Practical Limitations

6️⃣ *Hard to Scale*

- Increasing LSTM depth (many layers) leads to **unstable training** and **gradient issues** again.
- Transformers, by contrast, can scale up to hundreds of layers stably with residual connections and normalization.

7️⃣ *Difficult to Model Global Context*

- LSTMs process input **in order** (left-to-right or bidirectional), but they lack **direct connections** between distant tokens.
- This limits their ability to capture **global relationships** (e.g., between subject and verb far apart).

8️⃣ *Poor Interpretability*

- Internal gating dynamics are hard to interpret or visualize.
- Unlike attention maps in Transformers, you can’t easily see what the model is focusing on.
