

Long Short-Term Memory (LSTM)

Aayush Bajaj

July 10, 2026

Contents

1	the cell	1
2	the cell, drawn	2
3	why the additive path fixes vanishing gradients	2
4	peepholes and the gru	3
4.1	peephole connections	3
4.2	the gru	3
5	code: the cell from scratch	4
6	see also	5

the **vanilla rnn** cannot learn long-range dependencies: its gradient signal is a product of jacobians that shrinks or blows up geometrically with distance. the lstm’s answer is architectural, not numerical — give the network a second state, updated *additively* rather than by repeated matrix multiplication, and let learned gates decide what enters, what stays, and what leaves. the design dates to hochreiter and schmidhuber’s 1997 paper (*long short-term memory, neural computation* 9(8)), and for two decades it was simply what “rnn” meant in practice (Goodfellow, Ian, 2016).

hochreiter identified the vanishing-gradient problem in his 1991 diploma thesis, five years before the fix was published

1 the cell

one lstm step maps $(h_{t-1}, c_{t-1}, x_t) \mapsto (h_t, c_t)$. writing σ for the logistic sigmoid, $\odot$ for the elementwise product, and letting every weight matrix act on the concatenation $[h_{t-1}, x_t]$:

$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f)$	forget gate: what fraction of each cell coordinate survives
$i_t = \sigma(W_i[h_{t-1}, x_t] + b_i)$	input gate: how much new content is written
$o_t = \sigma(W_o[h_{t-1}, x_t] + b_o)$	output gate: how much of the cell is revealed
$g_t = \tanh(W_g[h_{t-1}, x_t] + b_g)$	candidate: the new content itself
$c_t = f_t \odot c_{t-1} + i_t \odot g_t$	cell state: gated carry plus gated write
$h_t = o_t \odot \tanh(c_t)$	hidden state: gated read-out

notation and design notes:

- the three gates are sigmoids, so their entries live in $(0, 1)$: they are soft, differentiable switches. the candidate is a tanh, living in $(-1, 1)$: it is *content*, not a switch.
- there are two states with different jobs. c_t is the long-term memory — a protected register bank that only ever gets scaled and added to. h_t is the working output — squashed, gated, and exposed to the rest of the network.
- parameter count: four matrices of shape $H \times (H + D)$ plus four biases — exactly four times the vanilla rnn cell. in implementations they are fused into one $4H \times (H + D)$ matrix and one matmul per step.
- a practical initialisation trick: set b_f to $+1$ (or higher), so the cell *remembers by default* at the start of training. an untrained forget gate sitting at $\sigma(0) = 0.5$ halves the memory at every step, which recreates the very problem the lstm exists to solve.

2 the cell, drawn

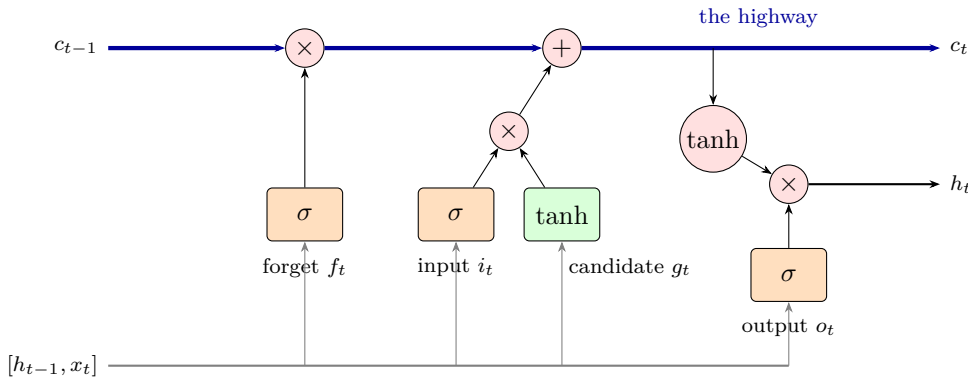


Figure 1: the lstm cell. the cell-state highway runs across the top: one elementwise scale, one addition — no matrix multiplication, no squashing. the three sigmoid gates below decide what is forgotten, written, and read.

3 why the additive path fixes vanishing gradients

compare the state-to-state jacobians of the two architectures. for the vanilla rnn,

$$\frac{\partial h_t}{\partial h_{t-1}} = \text{diag}(1 - h_t^{\odot 2}) W_h, \quad (1)$$

so credit travelling back k steps is squeezed through k matrix products and k tanh derivatives — geometric decay or explosion, as quantified on the [rnn page](#). for the lstm’s cell state, differentiate $c_t = f_t \odot c_{t-1} + i_t \odot g_t$ *holding the gates fixed* (their own dependence on h_{t-1} contributes additional, but separate, terms):

$$\frac{\partial c_t}{\partial c_{t-1}} = \text{diag}(f_t) \quad \implies \quad \frac{\partial c_t}{\partial c_k} = \text{diag}\left(\prod_{j=k+1}^t f_j\right). \quad (2)$$

no weight matrix. no activation derivative. the gradient along the cell path is a product of *learned, data-dependent* scalars per coordinate:

- where the network sets $f_j \approx 1$ (and writes nothing new), the gradient passes through untouched — hochreiter and schmidhuber called this loop the **constant error carousel**.
- where $f_j \approx 0$, the network has *chosen* to cut the gradient, because the past genuinely stopped mattering. vanishing becomes a decision, not a fate.
- explosion along the carousel is impossible: every factor satisfies $f_j < 1$ strictly. (the full gradient, including paths through the gates and h_t , can still explode — gradient clipping stays in the toolkit.)

the 1997 cell had no forget gate — the carousel weight was hard-wired to exactly 1. gers, schmidhuber and cummins added the forget gate in 1999 because never forgetting turned out to be its own pathology

the same additive-shortcut logic reappears across deep learning: resnet skip connections are the depth-wise version, highway networks the gated intermediate. whenever multiplication breaks gradients, the cure is addition.

4 peepholes and the gru

4.1 peephole connections

the standard cell decides its gates while blind to the very memory it is guarding: the gates see only $[h_{t-1}, x_t]$, and h_{t-1} is masked by the previous output gate. peephole connections (gers and schmidhuber, 2000) let each gate also read the raw cell state,

$$f_t = \sigma(W_f[h_{t-1}, x_t] + p_f \odot c_{t-1} + b_f), \quad (3)$$

and likewise for i_t (with c_{t-1}) and o_t (with the fresh c_t). they help on tasks needing precise timing and counting; large-scale comparisons find them mostly dispensable elsewhere.

4.2 the gru

the gated recurrent unit (cho et al., 2014) is the lstm’s ruthless minimalist cousin — two gates, one state, no output gate:

$$\begin{array}{ll} z_t = \sigma(W_z[h_{t-1}, x_t] + b_z) & \text{update gate} \\ r_t = \sigma(W_r[h_{t-1}, x_t] + b_r) & \text{reset gate} \\ \tilde{h}_t = \tanh(W_h[r_t \odot h_{t-1}, x_t] + b_h) & \text{candidate} \\ h_t = z_t \odot h_{t-1} + (1 - z_t) \odot \tilde{h}_t & \text{convex-combination update (cho et al.'s convention: } z_t \text{ keeps the past)} \end{array}$$

points of comparison:

- the gru’s z_t plays forget gate and input gate simultaneously, *coupled*: keep is exactly one-minus-write. the lstm can keep and write independently; the gru cannot.
- the additive highway survives — $\partial h_t / \partial h_{t-1}$ contains the term $\text{diag}(z_t)$, so the carousel argument goes through with z in the role of f .
- three matrices instead of four: 25% fewer parameters per unit of hidden width, one fewer state to carry.

- empirically the two trade wins task by task; neither dominates. the lstm remains the safer default when memory capacity matters (its cell can store unbounded magnitudes; the gru's state is trapped in $(-1, 1)$) (Goodfellow, Ian, 2016).

5 code: the cell from scratch

the forward pass, batched, with the shape and range invariants checked — plus a direct numerical verification of the carousel:

```

1 import numpy as np
2
3 rng = np.random.default_rng(7)
4
5 H, D, B = 8, 5, 4 # hidden, input, batch sizes
6 sigmoid = lambda z: 1.0 / (1.0 + np.exp(-z))
7
8 # one weight matrix per gate, acting on [h_{t-1}; x_t]
9 Wf, Wi, Wo, Wc = (rng.normal(0, 0.4, (H, H + D)) for _ in range(4))
10 bf = np.ones(H) # forget bias init to +1: remember by
   ↳ default
11 bi, bo, bc = (np.zeros(H) for _ in range(3))
12
13 def lstm_step(h, c, x):
14     z = np.concatenate([h, x], axis=1) # (B, H+D)
15     f = sigmoid(z @ Wf.T + bf) # forget gate (B, H) in (0,1)
16     i = sigmoid(z @ Wi.T + bi) # input gate (B, H) in (0,1)
17     o = sigmoid(z @ Wo.T + bo) # output gate (B, H) in (0,1)
18     g = np.tanh(z @ Wc.T + bc) # candidate (B, H) in (-1,1)
19     c = f * c + i * g # cell state: the additive highway
20     h = o * np.tanh(c) # hidden state
21     return h, c, (f, i, o, g)
22
23 h = np.zeros((B, H))
24 c = np.zeros((B, H))
25 for t in range(3):
26     x = rng.normal(0, 1, (B, D))
27     h, c, (f, i, o, g) = lstm_step(h, c, x)
28     print(f"t={t}: h {h.shape} in [{h.min():+.3f}, {h.max():+.3f}], "
29           f"c {c.shape} in [{c.min():+.3f}, {c.max():+.3f}]")
30
31 print("\ngate ranges at final step (all must sit inside their codomains):")
32 for name, gate, lo, hi in [("forget f", f, 0, 1), ("input i", i, 0, 1),
33                            ("output o", o, 0, 1), ("cand. g", g, -1, 1)]:
34     ok = (gate > lo).all() and (gate < hi).all()
35     print(f" {name}: min {gate.min():.3f} max {gate.max():.3f} "
36           f"inside ({lo},{hi}): {ok}")
37
38 # the constant error carousel: with f = 1, i = 0 the cell state is untouched
39 c_frozen = np.ones((B, H)) * 0.5
40 for _ in range(1000):
41     c_frozen = 1.0 * c_frozen + 0.0 * np.tanh(rng.normal(size=(B, H)))
42 print(f"\nafter 1000 steps with f=1, i=0: c unchanged? "
43       f"{np.allclose(c_frozen, 0.5)}")

```

```

t=0: h (4, 8) in [-0.354, +0.326], c (4, 8) in [-0.753, +0.615]
t=1: h (4, 8) in [-0.313, +0.287], c (4, 8) in [-0.625, +0.732]
t=2: h (4, 8) in [-0.550, +0.431], c (4, 8) in [-0.893, +0.968]

```

```

gate ranges at final step (all must sit inside their codomains):
forget f: min 0.457 max 0.893 inside (0,1): True
input i: min 0.206 max 0.909 inside (0,1): True

```

```
output o: min 0.229  max 0.771  inside (0,1): True
cand.  g: min -0.926  max 0.965  inside (-1,1): True
```

after 1000 steps with f=1, i=0: c unchanged? True

worth reading off the output:

- h_t is always strictly inside $(-1, 1)$ (it is a gated tanh), while c_t is *not* so confined — nothing squashes it after the additive update, and over long sequences it can grow without bound (the printed run is short, so its max of 0.968 merely hints at this): the register bank is genuinely unbounded.
- the forget-gate minimum (0.457) sits noticeably above the other gates' minima — the +1 bias at work.
- the final check is the carousel in its purest form: a thousand steps, zero drift, zero decay.

6 see also

- [recurrent neural networks](#) — the architecture this cell rescues, and bptt
- [sentiment analysis](#) — gated recurrence applied to text
- [transformers](#) — the architecture that eventually retired the lstm
- [feedforward deep neural networks](#) — where the residual version of the same trick lives

references

Goodfellow, Ian (2016). *Deep Learning*, MIT Press.