RNNs, LSTMs & GRUs

Lecture 10 · ES 667: Deep Learning

Prof. Nipun Batra
IIT Gandhinagar · Aug 2026

Learning outcomes

By the end of this lecture you will be able to:

1. Explain why an **MLP** fails on variable-length sequences.
2. Write a **vanilla RNN** cell in 4 lines of PyTorch.
3. Diagnose the **vanishing/exploding** gradient in **BPTT**.
4. Explain how **LSTM gates** sidestep vanishing gradients.
5. Contrast **LSTM vs GRU** and know when each is appropriate.
6. Name 2026 niches where RNNs (or Mamba/RWKV) still win.

Where we are

Module 5 opens. Until now everything was **feed-forward** — MLP, CNN. One input goes in, one output comes out, no memory.

Many real problems aren't like that:

- "He scored." — ambiguous without context.
- Audio, video, time series — ordered data.
- Language — tokens conditioned on all previous tokens.

> REFERENCE
>
> Today maps to **Bishop Ch 12** (RNNs). UDL skips RNNs and jumps to Transformers — we cover the recurrent ideas first because they motivate attention (L12).

Four questions

1. Why can't we just use an MLP for sequences?
2. What does a vanilla RNN actually compute?
3. Why do RNNs struggle with long-range dependencies — and how do LSTMs fix it?
4. When should you still use RNNs in 2026?

PART 1

Why MLPs fail on sequences

The parameter-sharing argument

The problem

Suppose you want to classify "He lives in Gandhinagar." as positive sentiment.

If you feed this to an MLP:

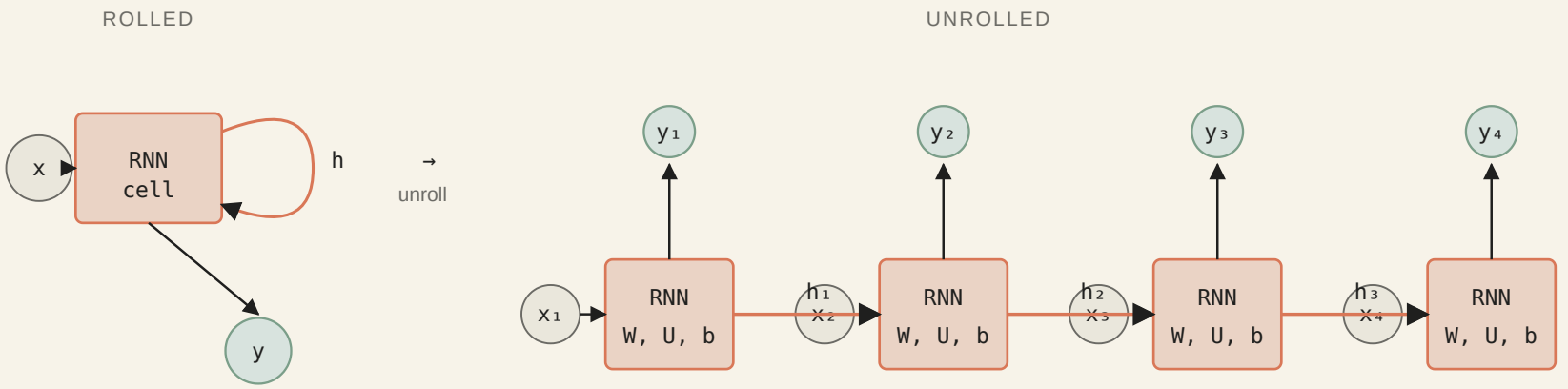
- Words have no inherent order — must one-hot the position: `(token, position)`.
- Vocabulary is 50k words × max length 100 → input dim = 5,000,000.
- No reuse: the word "Gandhinagar" at position 5 is a completely different feature from the same word at position 23.

> KEY IDEA
>
> An MLP has **no inductive bias for time** — it would need to relearn what each word means, once per position.

Weight sharing across time · the RNN trick

Key idea: the same network processes each timestep, with a *memory* that carries across.

RNN unrolled in time — same cell, same weights, different timestep



$$h_t = \tanh(W \cdot x_t + U \cdot h_{t-1} + b) \cdot y_t = V \cdot h_t$$

Same W, U, b at every t — parameter sharing across time is the RNN inductive bias.

RNN · step-by-step on "I love deep learning"

Tiny model · input dim $d_{\text{in}} = 2$, hidden dim $d_h = 3$. $h_0 = [0, 0, 0]^\top$. Embeddings (toy): $x_1 = [1, 0]^\top$ (I), $x_2 = [0, 1]^\top$ (love), $x_3 = [1, 1]^\top$ (deep), $x_4 = [0, 0]^\top$ (learning).

Weights:

$$W = \begin{pmatrix} 0.1 & 0.2 \\ 0.3 & 0.4 \\ 0.5 & 0.6 \end{pmatrix}, \quad U = \begin{pmatrix} 0.1 & 0.2 & 0.3 \\ 0.4 & 0.5 & 0.6 \\ 0.7 & 0.8 & 0.9 \end{pmatrix}$$

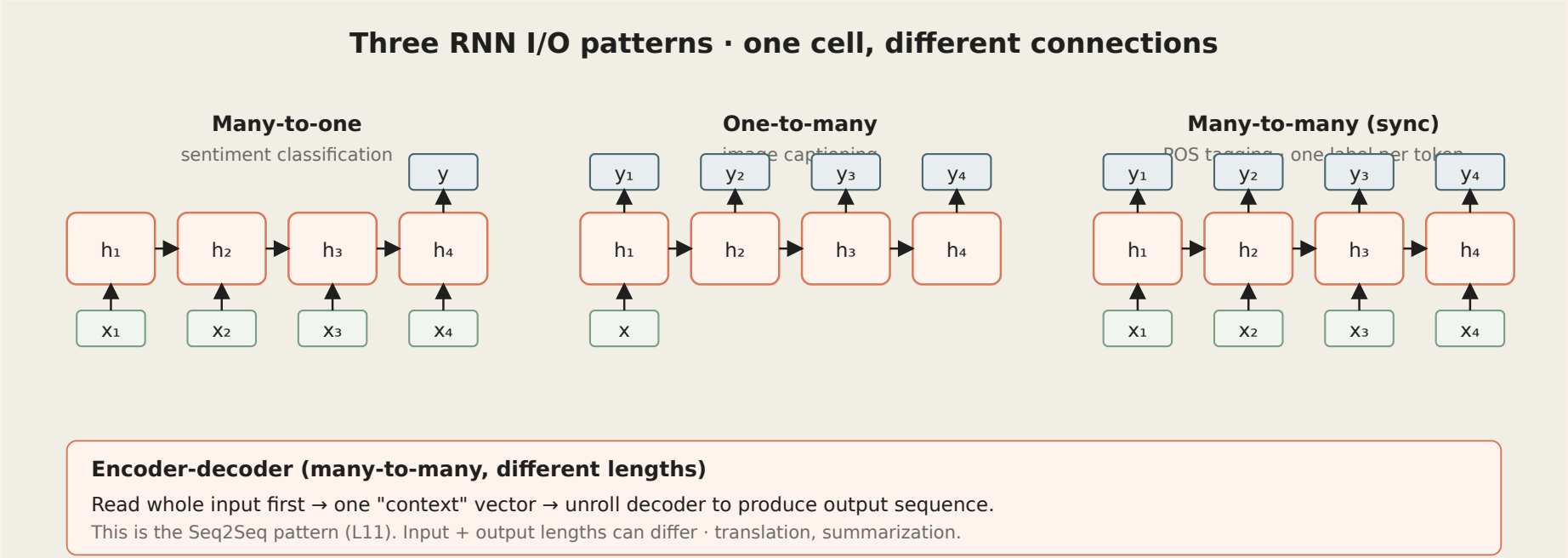
Step 1 · "I". $h_0 = 0$, so $h_1 = \tanh(W x_1) = \tanh([0.1, 0.3, 0.5]^\top) \approx [0.099, 0.291, 0.462]^\top$. This vector summarizes the sequence so far.

Step 2 · "love". Use the **same** W, U with new input x_2 and previous state h_1: $h_2 = \tanh(W x_2 + U h_1)$.

The recurrence:

W, U are **shared** across steps — unlike an MLP that would learn a different `(token, position)` feature for every slot.

RNN I/O patterns · three shapes



Encoder-decoder (many-to-many, different lengths)

Read whole input first → one "context" vector → unroll decoder to produce output sequence.

This is the Seq2Seq pattern (L11). Input + output lengths can differ · translation, summarization.

The three RNN use-patterns

Many-to-one

- Input: sequence
- Output: one label at the end
- e.g., sentiment classification

One-to-many

- Input: one feature vector
- Output: a sequence
- e.g., image captioning

Many-to-many (in sync) · e.g., POS tagging — one label per input token.
Many-to-many (encoder-decoder) · e.g., translation — input sequence, output sequence, different lengths. This is the Seq2Seq pattern covered in L11.

Four shapes, same cell.

RNN in PyTorch · by hand

```
class RNNCell(nn.Module):
    def __init__(self, d_in, d_h):
        super().__init__()
        self.W = nn.Linear(d_in, d_h, bias=False)
        self.U = nn.Linear(d_h,  d_h, bias=True)
        # tanh folds the output back to [-1, 1]

    def forward(self, x_t, h_prev):
        return torch.tanh(self.W(x_t) + self.U(h_prev))

# Unrolled loop
h = torch.zeros(batch, d_h)
for t in range(seq_len):
    h = cell(x[:, t], h)
```

`nn.RNN`, `nn.LSTM`, `nn.GRU` handle the loop + CUDA kernels for you.

PART 2

BPTT and vanishing gradients in time

Same problem as depth, now along the time axis

BPTT · the telephone game

INTUITION

Analogy · Chinese whispers. Tell a secret to person 1. They whisper to person 2, etc. By person 20, the message is garbled (vanished) or wildly distorted (exploded).

The gradient is that secret. It travels backward from the loss to the start of the sequence, and at each step it gets transformed.

BPTT · derive the gradient product

Start with the simplest possible RNN: $h_t = W h_{t-1}$ (no input, no $\tanh$). Unroll 3 steps:

- $h_1 = W h_0$
- $h_2 = W^2 h_0$
- $h_3 = W^3 h_0$

Chain rule:

$$\frac{\partial h_3}{\partial h_0} = \underbrace{\frac{\partial h_3}{\partial h_2}}_{=W} \cdot \underbrace{\frac{\partial h_2}{\partial h_1}}_{=W} \cdot \underbrace{\frac{\partial h_1}{\partial h_0}}_{=W} = W^3$$

For sequence length T, gradient $\propto W^T$.

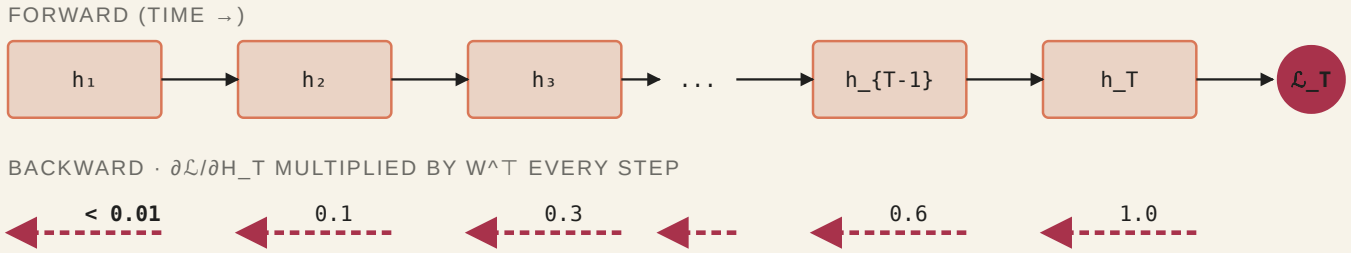
Add $\tanh$ back: $h_t = \tanh(W h_{t-1})$. Each Jacobian is $W^\top \cdot \mathrm{diag}(\tanh'(\cdot))$. Since $\tanh' \in (0, 1)$:

$$\frac{\partial \mathcal{L}}{\partial h_1} = \frac{\partial \mathcal{L}}{\partial h_T} \cdot \prod_{t=2}^{T} \frac{\partial h_t}{\partial h_{t-1}}$$

A product of T scaled-down matrices · vanishing if $\|W\| < 1$, exploding if $\|W\| > 1$.

Vanishing gradients in time

BPTT — gradient travels back through every timestep, multiplied every step



$$\partial \mathcal{L}/\partial h_1 = \prod (\partial h_t/\partial h_{t-1}) \cdot \partial \mathcal{L}/\partial h_T = \prod W^\top \cdot \mathrm{diag}(\tanh'(\cdot)) \cdot \partial \mathcal{L}/\partial h_T$$

if largest eigenvalue of $W^\top$ is

$< 1 \rightarrow$ product $\rightarrow 0$ (vanishing) · $> 1 \rightarrow$ product $\rightarrow \infty$ (exploding)

Long-range dependencies (> 20 steps) are the pain point vanilla RNNs cannot handle. This is what LSTM gates fix.

Worked example · BPTT on 3 timesteps

DERIVATION

Consider a tiny RNN with scalar state · $h_t = \tanh(wh_{t-1} + x_t) \cdot w = 0.5 \cdot \tanh'(\cdot) \leq 1.$

$$\frac{\partial h_3}{\partial h_0} = \underbrace{w \tanh'(\cdot)}_{\leq 0.5} \cdot \underbrace{w \tanh'(\cdot)}_{\leq 0.5} \cdot \underbrace{w \tanh'(\cdot)}_{\leq 0.5} \approx 0.125$$

- 3 steps · gradient at most 0.125 (already 8× smaller)
- 10 steps · $\leq 0.5^{10} \approx 10^{-3}$
- 50 steps · $\leq 0.5^{50} \approx 10^{-15}$

That's why vanilla RNNs can't learn dependencies across more than ~20 timesteps. Every step in the product pulls the gradient toward zero if $|w \tanh'| < 1$, or toward infinity if > 1. LSTMs (next) sidestep this via additive gated updates.

Truncated BPTT · the practical fix

For long sequences (thousands of steps), full BPTT is expensive.

Truncated BPTT (TBPTT) — only backpropagate K steps at a time:

```
for chunk in sequence.split(K, dim=1):
    h_detached = h.detach()              # cut gradient here
    for t in range(chunk.size(1)):
        h = cell(chunk[:, t], h)
    loss = criterion(h, target)
    loss.backward()
    opt.step()
```

Typical $K = 32$ to 256. Keeps training tractable at the cost of losing very-long-range gradient signal.

Gradient clipping · the second fix

Exploding gradients are worse than vanishing — one bad step can destroy weeks of training. Clip by global norm:

```
torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)
opt.step()
```

KEY IDEA

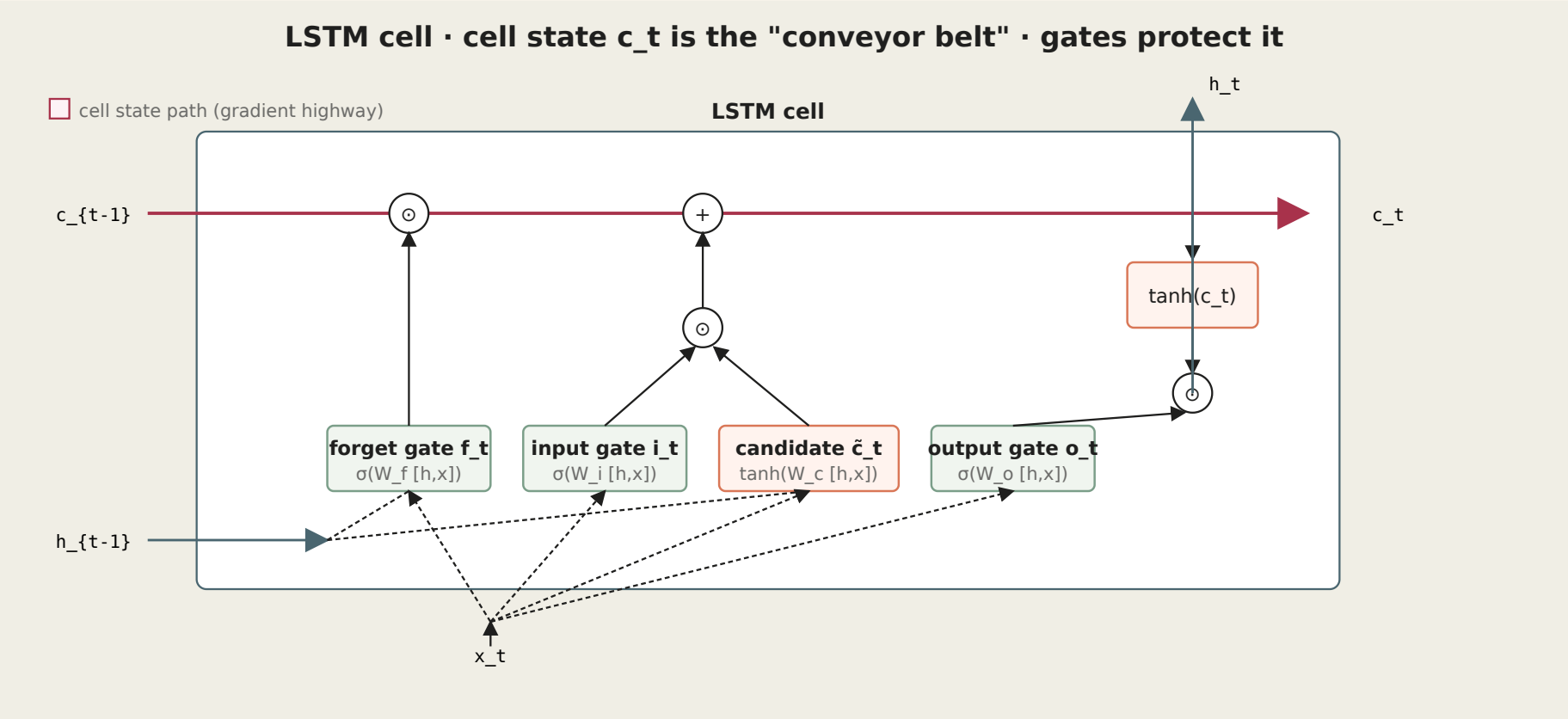
Pascanu et al. 2013 showed clipping at norm ~1 makes RNN training robust. Still the default for any sequence model — RNN, LSTM, Transformer. Cheap insurance against numerical catastrophe.

PART 3

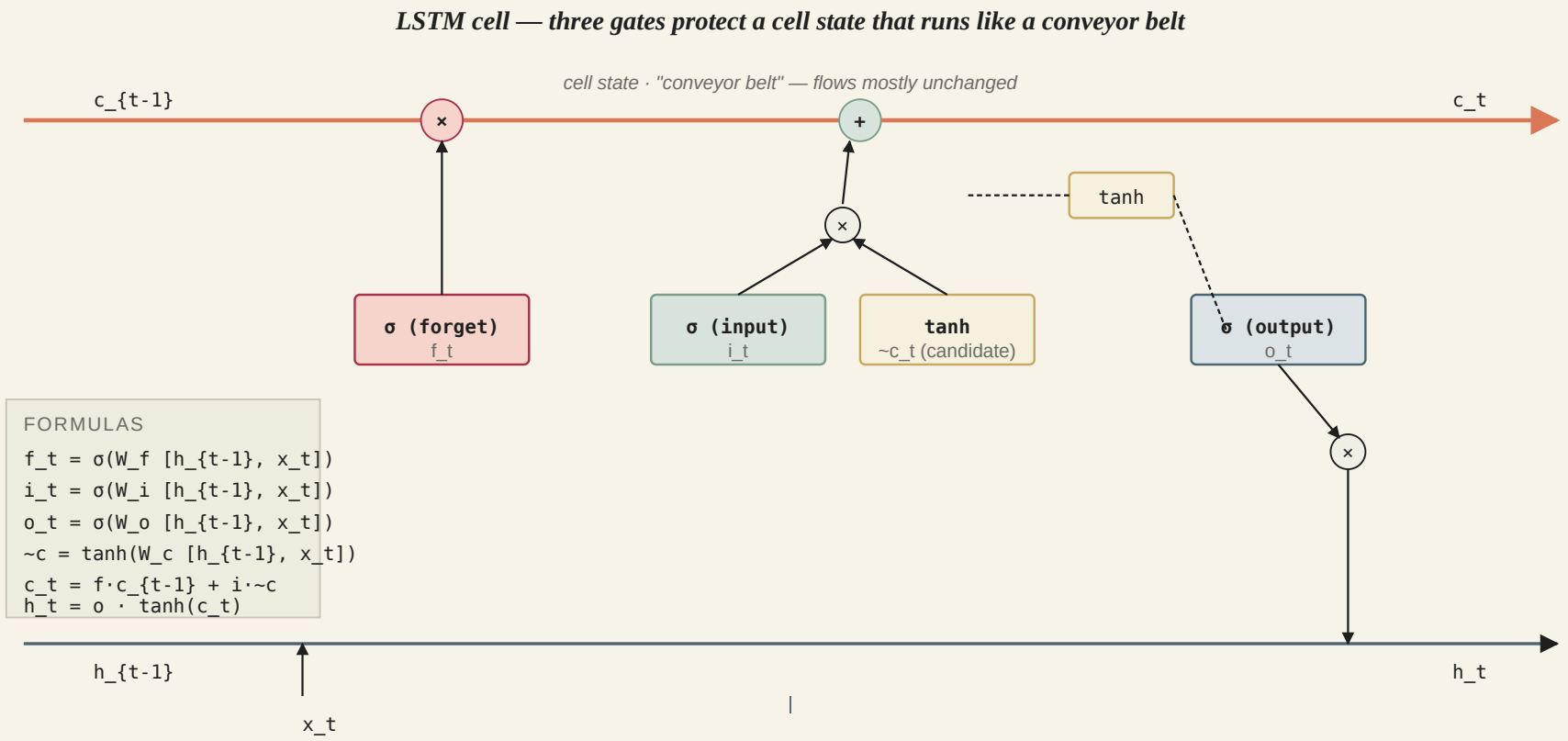
LSTM · the gating fix

Three sigmoid gates protect a cell state

LSTM cell · annotated



LSTM cell architecture



IN PRACTICE

Interactive: drag forget/input/output sliders; see the cell state freeze, flow, or reset — lstm-gates.

LSTM · gatekeeper, janitor, press secretary

KEY IDEA

A vanilla RNN treats every input the same. The LSTM has three "specialists":

- **Gatekeeper** (input gate) · is this new word important enough to write to memory?
- **Janitor** (forget gate) · cleans out old memories that don't matter anymore.
- **Press secretary** (output gate) · decides which parts of internal memory to expose to downstream layers.

The next slide gives the math · keep these roles in mind as you read the equations.

LSTM · the conveyor-belt analogy

INTUITION

A vanilla RNN crams everything into one memory h_t — like doing complex calculations using only 8 CPU registers; things get overwritten constantly.

LSTM adds a separate **memory conveyor belt** · the **cell state** $\mathbf{c}_t$. Information travels along this belt unchanged unless special **gates** intervene:

- **Forget gate** · the janitor — removes items from the belt.
- **Input gate** · the gatekeeper — decides what new items to add.
- **Output gate** · the press secretary — picks what to show outside.

A protected long-term memory + learned controllers for write / forget / read. That's the whole idea.

LSTM · build the equations step-by-step

Combine inputs into one **control vector** $[\mathbf{h}_{t-1}, \mathbf{x}_t]$. Use it to drive every gate.

Step 1 · Forget gate (sigmoid → $[0, 1]$):

$$\mathbf{f}_t = \sigma(W_f[\mathbf{h}_{t-1}, \mathbf{x}_t] + b_f)$$

0 → forget · 1 → keep.

Step 2 · Input gate + candidate:

$$\mathbf{i}_t = \sigma(W_i[\mathbf{h}_{t-1}, \mathbf{x}_t] + b_i), \quad \tilde{\mathbf{c}}_t = \tanh(W_c[\mathbf{h}_{t-1}, \mathbf{x}_t] + b_c)$$

$\mathbf{i}_t$ controls *how much* of the candidate $\tilde{\mathbf{c}}_t$ to write.

Step 3 · Update the cell state · throw out old, add new:

$$\mathbf{c}_t = \underbrace{\mathbf{f}_t \odot \mathbf{c}_{t-1}}_{\text{kept}} + \underbrace{\mathbf{i}_t \odot \tilde{\mathbf{c}}_t}_{\text{added}}$$

The crucial **+** is what makes gradients flow.

Step 4 · Output gate + hidden state:

Worked numeric · single-neuron LSTM

1D state. Setup:

- $c_{t-1} = 5.0$ ("the subject is plural")
- $h_{t-1} = 0.9$, new input $x_t = 0.2$ (the word "was")

Network has learned (for this input):

- **Forget** $f_t = 0.1$ — sees "was" → forget the plural memory.
- **Input** $i_t = 0.9$ — write new info aggressively.
- **Candidate** $\tilde{c}_t = -2.0$ — encodes "subject is singular".

Compute new cell state:

$$c_t = f_t \cdot c_{t-1} + i_t \cdot \tilde{c}_t = 0.1 \cdot 5.0 + 0.9 \cdot (-2.0) = 0.5 - 1.8 = \mathbf{-1.3}$$

The memory **flipped** from large positive (plural) to negative (singular) in one step — exactly because the forget gate was small *and* the input gate was large.

Each gate · in plain English

- **Forget gate $\mathbf{f}_t$** · "what should I erase from the cell state?"
  - Close to 0 · forget (reset a counter, flush old context)
  - Close to 1 · keep it around (persistent memory)
- **Input gate $\mathbf{i}_t$** · "how much of the new candidate should I actually write?"
  - Close to 0 · ignore this input
  - Close to 1 · accept fully
- **Output gate $\mathbf{o}_t$** · "what of the cell state do I expose to downstream layers?"
  - Close to 0 · keep memory silent
  - Close to 1 · project it out

KEY IDEA

The LSTM's "memory" is the cell state $\mathbf{c}_t$; the gates are **learned controllers** that decide when to write, keep, or read. Think of it as a differentiable tiny memory cell plus a learned read/write scheduler.

Why gating fixes vanishing gradients

Vanilla RNN. $h_t = \tanh(Wh_{t-1} + \cdots)$. Backward Jacobian:

$$\frac{\partial h_t}{\partial h_{t-1}} = W^\top \cdot \mathrm{diag}(\tanh'(\cdot))$$

A **matrix multiplication** at every step → product of matrices → vanishing/exploding.

LSTM cell state. $c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$. Along the cell-state path:

$$\frac{\partial c_t}{\partial c_{t-1}} = f_t$$

That's it — **no matrix multiplication**. Just element-wise multiplication by the forget gate.

If the network learns $f_t \approx 1$ ("don't forget"), the gradient flows through unchanged. The "gradient highway" works in time the same way ResNet's identity skip works in depth.

Worked numeric · gradient flow over 100 steps

Suppose memory must be preserved → forget gate trained to $f_t = 0.99$ consistently.

	AFTER 100 STEPS
LSTM along cell state	$0.99^{100} \approx \mathbf{0.366}$ — small but nonzero
Vanilla RNN (optimistic factor 0.5)	$0.5^{100} \approx 7.9 \times 10^{-31}$ — completely vanished

LSTM signal **survives**. RNN signal is below floating-point precision.

This is the same idea as ResNet skip connections, in the time dimension.

PART 4

GRU · the lighter sibling

Fewer gates, comparable accuracy

GRU · a simpler, smarter gate

Two design questions Cho et al. asked in 2014:

1. Do we really need a *private* cell state c_t and a *public* hidden state h_t? Just keep one.
2. The LSTM's forget and input gates are often opposites. Can we use one gate that picks "old vs new"?

Result · the **update gate $\mathbf{z}_t$** is a dial:

- $\mathbf{z}_t \approx 0 \rightarrow$ keep the old state.
- $\mathbf{z}_t \approx 1 \rightarrow$ switch to the new candidate.

GRU · build the equations step-by-step

Step 1 · Reset gate. How much of the past to use when forming the candidate.

$$\mathbf{r}_t = \sigma(W_r[\mathbf{h}_{t-1}, \mathbf{x}_t])$$

Step 2 · Candidate. Reset gate filters the past first.

$$\tilde{\mathbf{h}}_t = \tanh(W[\mathbf{r}_t \odot \mathbf{h}_{t-1}, \mathbf{x}_t])$$

Step 3 · Update gate. How much new vs old to use.

$$\mathbf{z}_t = \sigma(W_z[\mathbf{h}_{t-1}, \mathbf{x}_t])$$

Step 4 · Final state · linear interpolation.

$$\mathbf{h}_t = (1 - \mathbf{z}_t) \odot \mathbf{h}_{t-1} + \mathbf{z}_t \odot \tilde{\mathbf{h}}_t$$

The **additive** structure (like LSTM's cell state) is what keeps gradients flowing.

Worked numeric · scalar GRU

1D state. $h_{t-1} = 0.8$ ("expecting a verb"). Candidate $\tilde{h}_t = -0.5$ ("new word is a noun").

Case 1 · $z_t = 0.1$ (keep old).
$h_t = 0.9 \cdot 0.8 + 0.1 \cdot (-0.5) = 0.72 - 0.05 = \mathbf{0.67}$
Close to old state — input mostly ignored.

Case 2 · $z_t = 0.9$ (use new).
$h_t = 0.1 \cdot 0.8 + 0.9 \cdot (-0.5) = 0.08 - 0.45 = \mathbf{-0.37}$
Close to candidate — state nearly fully replaced.

The single update gate gives the network smooth control between "preserve" and "overwrite."

LSTM vs GRU · a history sentence

In the late 2010s, ML papers often included an ablation · "we tried LSTM and GRU and picked whichever worked." By 2019, most groups defaulted to whichever they had better library support for.

INTUITION

That casualness told the story · **the gating trick matters; which gates you pick doesn't much.** Any additive-gated recurrence works; the architectural variants are micro-optimizations on the core idea.

LSTM vs GRU · when to pick which

	LSTM	GRU
Gates	3 + candidate	2 + candidate
State	cell + hidden	hidden only
Params (d_h = 128)	4 · 128 · 256 = 131k	3 · 128 · 256 = 98k
Accuracy	baseline	often tied
Training speed	slower	~15% faster

INTUITION

Empirically close — both far beyond vanilla RNNs on long-range tasks. GRU was often preferred pre-Transformer; today either is fine for the few remaining RNN use cases.

Bidirectional + stacked RNNs

Bidirectional · run one RNN left-to-right and another right-to-left; concatenate the outputs. Used for classification/tagging (not generation).

```
self.rnn = nn.LSTM(d_in, d_h, num_layers=2, bidirectional=True, batch_first=True)
```

Stacked (deep) RNN · layer ℓ's hidden state feeds layer $\ell + 1$. Each layer captures different abstraction.

Both tricks combinable: 2-layer bidirectional LSTMs were the standard NLP architecture from 2015–2017 (before Transformers).

PART 5

When to still use RNNs in 2026

The 2026 reality

Transformers have largely replaced RNNs for:

- **Language modeling** (GPT, Llama, Claude)
- **Machine translation** (Google Translate, DeepL)
- **Speech recognition** (Whisper)
- **Music generation** (MusicLM)

But RNNs are still the right choice for:

Streaming / online inference

- Process input one token at a time
- O(1) state update
- No need to recompute attention over history

Tiny devices

- Microcontrollers, always-on sensors
- Memory budget in KB
- LSTM cell: ~1k params

RWKV and Mamba · the RNN comeback

Starting in 2023, a new class of models has re-emerged · **state-space models** and **linear RNNs** that match Transformer quality with **O(1) inference per token**.

DERIVATION

- **RWKV** (Peng 2023) · linear attention reformulated as RNN · trained in parallel, runs as recurrent at inference.
- **Mamba** (Gu 2023) · selective state-space model · same asymptotic scaling, competitive on language.
- **Mamba-2** (2024) · faster, matches Transformer-7B quality.

INTUITION

The story isn't "RNNs are dead" — it's "vanilla RNNs with sequential gradients couldn't scale." Modern parallelizable RNNs are a quiet comeback. Watch this space.

A preview · the problem RNNs can't solve

Consider translating:

> *"The animal didn't cross the street because it was too tired."*

To translate "it" correctly, the model must look back to "animal" — maybe 6 tokens ago.

An RNN compresses all of that into a single h_t vector. Longer sentences → more to compress → more is lost.

KEY IDEA

The next lecture (L11) examines encoder-decoder Seq2Seq, which also struggles with this **fixed-length bottleneck**. That struggle motivates **attention** (L12) — the idea that finally let sequence models scale.

Summary · Lecture 10 — summary

- **MLPs fail on sequences** because they can't share parameters across time.
- **RNN** — same cell, shared weights, hidden state carries memory forward.
- **BPTT** — backprop through unrolled graph; same vanishing/exploding problem as depth.
- **LSTM** — gated cell state is an additive "conveyor belt"; gradients flow through gates, not through tanh products.
- **GRU** — simpler (2 gates instead of 3); often equivalent accuracy, ~15% faster.
- **2026** — Transformers own most sequence tasks, but RNNs still win for streaming and tiny devices.

Read before Lecture 11

Bishop Ch 12 · Seq2Seq.

Next lecture

Seq2Seq + the motivation for attention — encoder-decoder architecture, teacher forcing, beam search, and the bottleneck that made attention inevitable.

NOTEBOOK

Notebook 10 · `10-lstm-from-scratch.ipynb` — implement an LSTMCell with only `nn.Linear` layers; verify output matches `nn.LSTMCell`; train a char-level LSTM on Tiny Shakespeare.