DEEP LEARNING

Recurrent networks

RNNs, LSTMs & GRUs: from spatial to temporal structure

Prerequisites: CNNs · Embeddings · Transfer learning

Dr. León Felipe Palafox Novack

lfpalafox@up.edu.mx

01 Why sequences

02 The recurrence

03 Vanishing & exploding signals

04 LSTM: gated memory

05 GRU: two gates

06 RNN vs LSTM vs GRU

07 In practice

08 What came next

09 Worked example

02

What your toolkit can't do yet.

FIXED-SIZE INPUT SLOT · CAPACITY N = 3



With 300-dimensional word embeddings, the 3-token slot is a vector of exactly **900** inputs. A 9-token sentence is **2,700** numbers: a different input size, so the same network cannot even read it.

Cutting to 3 tokens throws information away; padding every sentence to the longest one wastes inputs and still fails on the next, longer sentence.

MLP

Needs a **fixed-size input**. A sequence has no fixed length.

CNN

Fixed receptive field. Sees a local window, not arbitrary-distance history.

A 1-D convolution with kernel 3 sees 3 neighbouring tokens; L stacked layers see 2L + 1.

Both

No persistent state. No notion of "what came before".

Each input is processed as if it were the first one.

03

Sequences are everywhere.

NATURAL LANGUAGE

the | cat | sat | on | the | mat

one embedding x_t per token · 6 tokens here

SPEECH / AUDIO



16,000 samples per second (dummy signal)

MUSIC



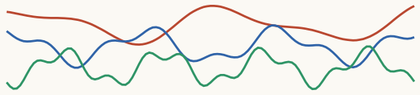
notes over time (dummy melody)

VIDEO FRAMES



25–30 frames per second

SENSOR / IOT STREAMS



3-axis accelerometer (dummy signal)

DNA / PROTEIN

GAGGCTATGGACCTCTGCATACACTAATC
CTCCATGTGTGAGGCTGATCCTAATCT

4-letter alphabet · up to millions of bases (random)

TIME SERIES



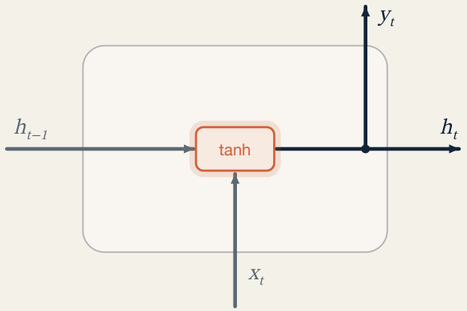
one value per day, weekly cycle (dummy)

A SOLUTION MUST 1 handle **variable length** 2 respect **order** 3 carry **memory** across arbitrary distance

This checklist is our rubric for the plain RNN, then the LSTM and the GRU.

04

The recurrence: one cell, reused over time.



$$h_t = \tanh(W_{hh} \cdot h_{t-1} + W_{xh} \cdot x_t + b_h)$$

$$y_t = g(W_{hy} \cdot h_t)$$

> **h_t** is a compressed, lossy summary of everything seen so far, not a memory buffer.

x_t input · **h_t** hidden state · **y_t** output · **W** weights (shared by every step)

A TINY RNN WITH RANDOM WEIGHTS READS “THE CAT SAT ON THE ___”

	the	cat	sat	on	the	___
$h[1]$	0.38	0.90	−0.41	0.69	0.34	
$h[2]$	0.30	−0.33	−0.34	−0.20	0.33	
$h[3]$	0.14	0.20	−0.61	−0.05	0.04	
$h[4]$	−0.58	0.41	−0.18	0.44	−0.49	
$h[5]$	0.01	0.12	−0.51	0.37	−0.12	
$h[6]$	−0.25	−0.95	−0.41	−0.70	−0.18	

NEXT WORD?

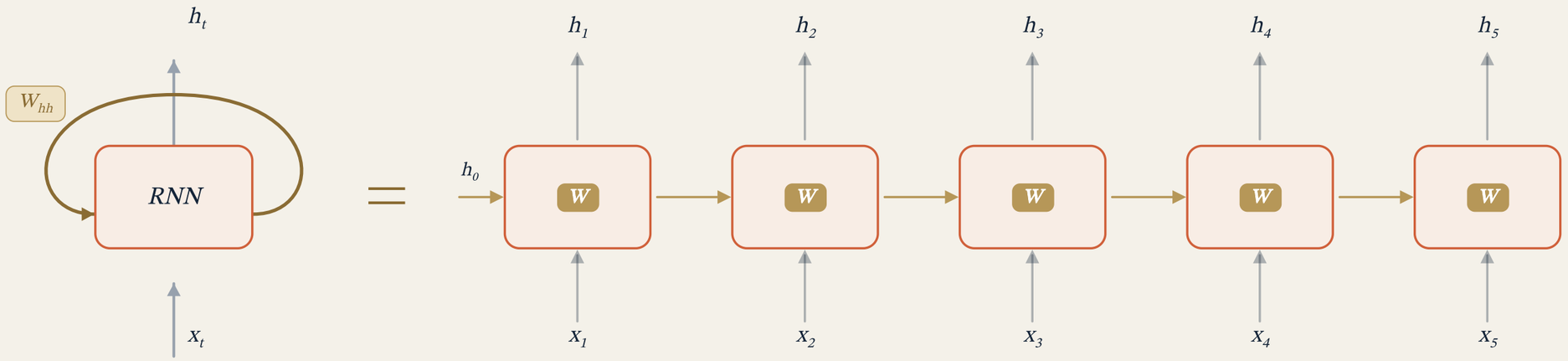
the	0.17
cat	0.16
sat	0.20
on	0.13
mat	0.15
dog	0.19

Real numbers, meaningless model: 6 hidden units, 4-number dummy embeddings and PyTorch-style random initialisation. Before training, the next-word guess is close to uniform (1/6 ≈ 0.17): **p(mat) = 0.15**. Training would push it up.

t = 1 · the | t = 2 · cat | t = 3 · sat | t = 4 · on | **t = 5 · the**

05

Unrolling: the same network, drawn in time.



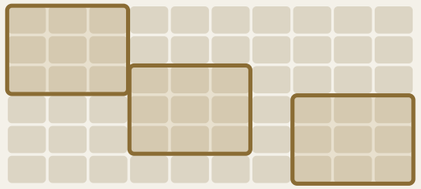
One network drawn across timesteps, **not many networks**. The same **W** is reused at every step.

In the tiny RNN of the previous slide that is **24** numbers in W_{xh} (6×4) + **36** in W_{hh} (6×6) + **6** in b_h = **66** numbers, reused at t = 1, 2, 3, 4, 5 (and for any longer sentence).

06

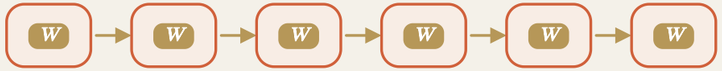
Sharing across time = the CNN trick, one axis over.

CNN · SHARING ACROSS SPACE



The **same 3×3 filter** at every position of the image.

RNN · SHARING ACROSS TIME



The **same W** at every step: any sequence length, and what is learned at one position transfers to the others.

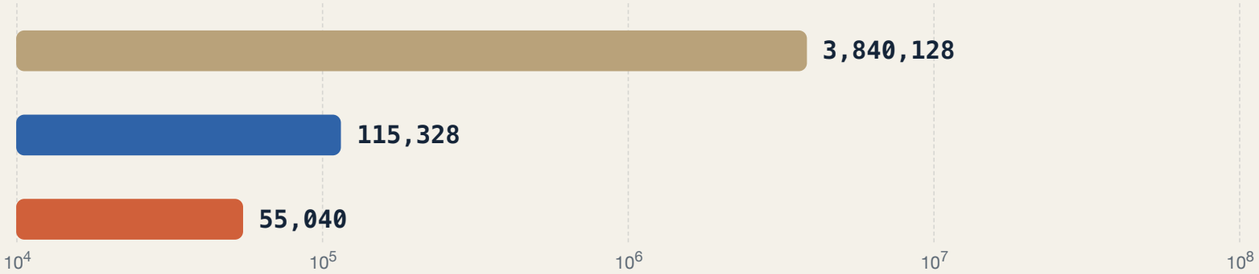
WEIGHT SHARING ALONG A STRUCTURED AXIS

PARAMETERS TO READ T WORD VECTORS (300-D) INTO 128 UNITS

MLP on the flattened sequence
`T·300·128 + 128`

Conv1d, kernel 3
`3·300·128 + 128`

RNN (nn.RNN)
`300·128 + 128·128 + 2·128`



SEQUENCE LENGTH: T = 5 | T = 20 | **T = 100** | T = 1000

Real counts (PyTorch 2.15). The MLP grows with T and only accepts one length; the convolution and the RNN do not grow. Training: the same backpropagation you know, run over the unrolled network (**BPTT**). We won't go deeper than that today.

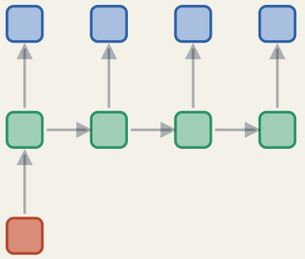
07

Recurrence is flexible about input/output shape.

1

one-to-many

image captioning



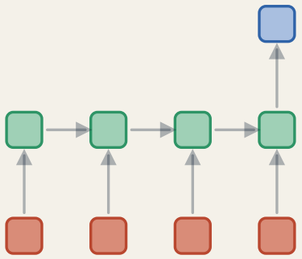
`image feature (32, 2048)`
→ h_0 `(32, 128)`
→ `12 steps` → `(32, 12, 5000)`

32 captions × 12 words × 5,000-word vocabulary

2

many-to-one

sentiment



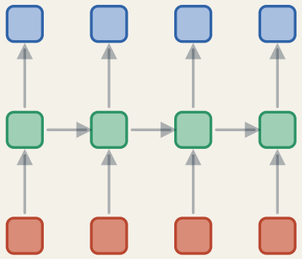
`x (32, 20, 300)`
→ h_n `(1, 32, 128)`
→ `logits (32, 2)`

32 reviews × {negative, positive}

3

many-to-many, aligned

POS tagging



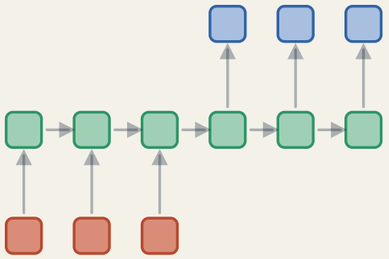
`x (32, 20, 300)`
→ `output (32, 20, 128)`
→ `tags (32, 20, 17)`

one of 17 universal POS tags per word

4

many-to-many, seq2seq

translation → **next session**



`source (32, 20, 300)`
→ (h_n, c_n) `(1, 32, 128)`
→ `target logits (32, 15, 5000)`

the encoder's last state starts the decoder

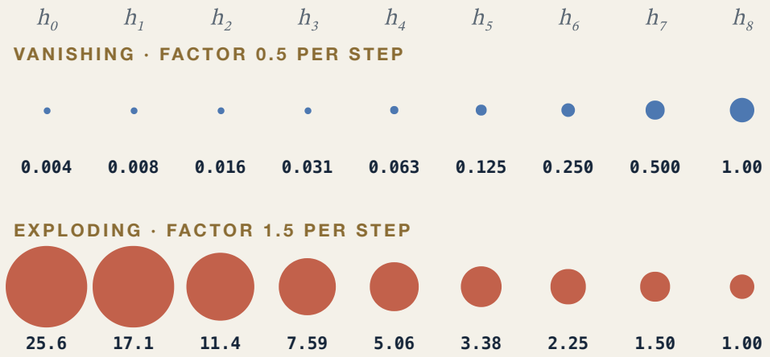
x_t input h_t RNN state y_t output The x_t are exactly the embeddings you already use. Shapes: real PyTorch outputs, batch-first, random weights.

08

The catch: the learning signal must travel back through time.

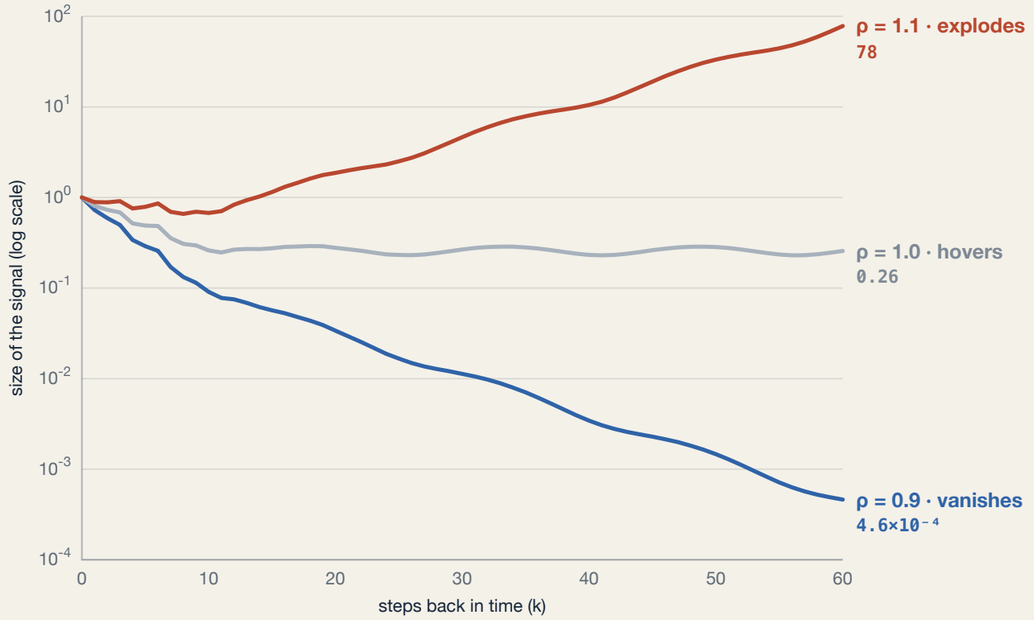
To learn that an early token matters for a late output, the learning signal travels **back across many steps**.

At every step it is multiplied again by the same recurrent weights, so it tends to **shrink toward zero** (vanishing) or **blow up** (exploding).



The signal starts at h_8 with size 1 and is multiplied once per step on its way back to h_0.

SAME MATRIX, K TIMES: RANDOM 32×32 W



ρ is the spectral radius (largest |eigenvalue|) of W. Forward computation only: v ← W^Tv, sixty times, from a random unit vector.

09

Two failure modes — one cheap fix, one deep problem.

Exploding

Loss spikes to NaN. **Visible.** Fixed pragmatically with **gradient clipping**.

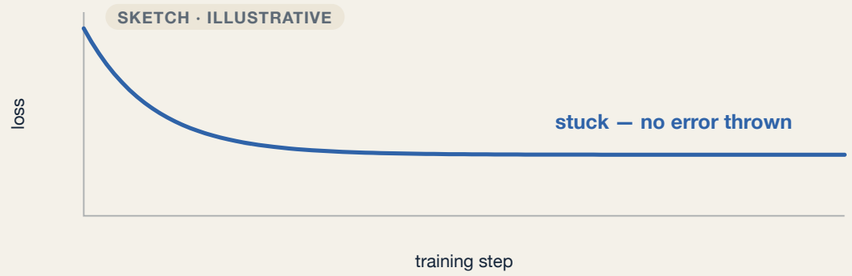


Real: clip_grad_norm_(params, max_norm = 5)

`g = (30, −40) · ‖g‖ = 50 > 5 → g × 5/50 = (3, −4)`

Vanishing

Silent. No error: the model just plateaus on long-range dependencies. Needs an **architectural** answer.



A factor per step, fifty times

$0.9^{50} = 0.005$ (half a percent left) · $1.1^{50} = 117$

Clipping handles explosion. Vanishing needs a new architecture →

10

LSTM: add a memory highway.

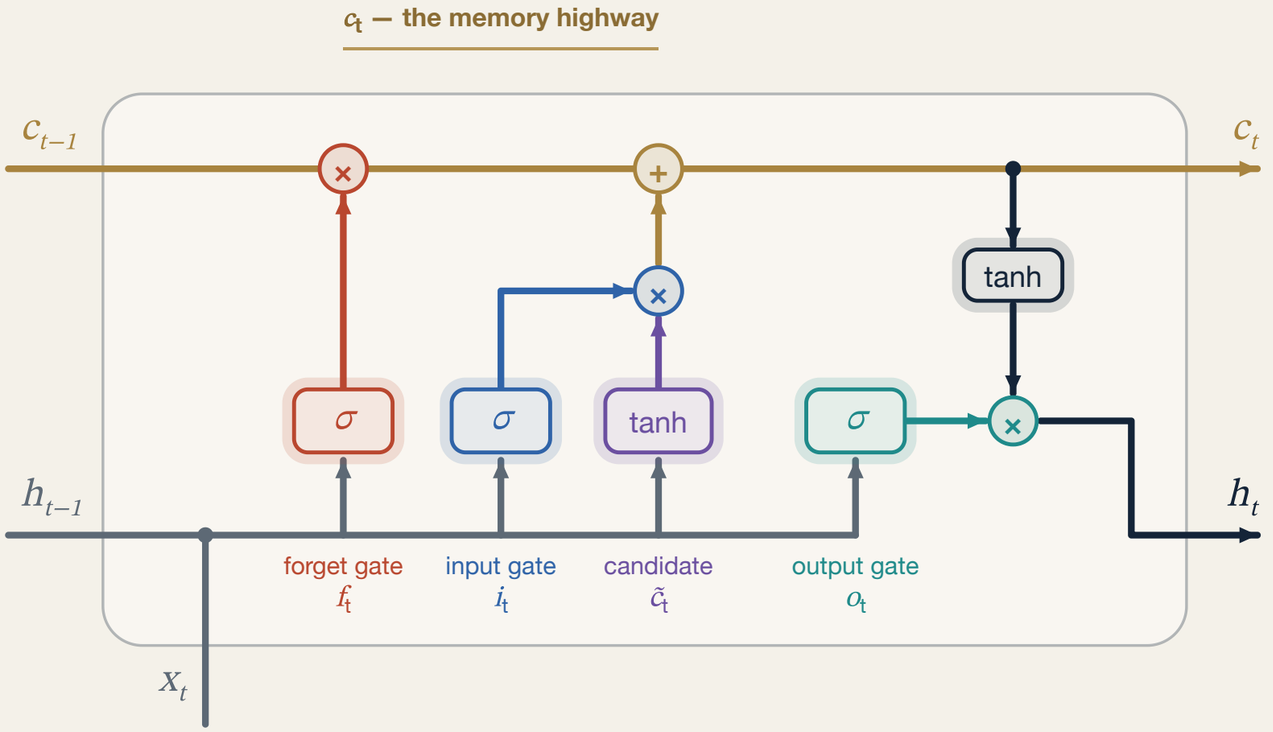
Give the network a separate **cell state c_t** with an **additive** update: a highway the signal travels without being repeatedly squashed.

NOTATION

x_t input · h_t hidden state · c_t cell state · y_t output

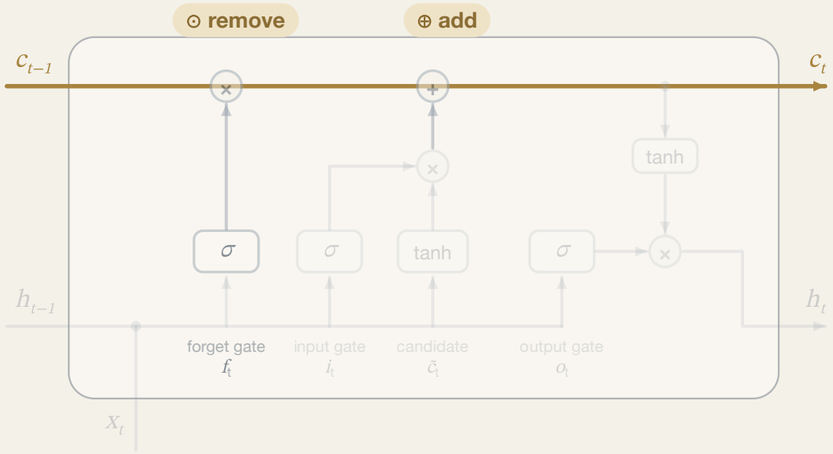
CONVENTION

time flows left → right · inputs enter from below · outputs leave above · c_t travels on the top highway



11

The cell state: a lightly-touched conveyor belt.



c_t runs straight through, touched by just **one** ⊙ (remove) and **one** ⊕ (add). Minimal interference is the point.

A plain RNN **overwrites its whole state** through a nonlinearity at every step; this doesn't.

STORED VALUES WHILE READING "THE MOVIE WAS NOT GOOD"

	the	movie	was	not	good
Plain RNN · h_t					
Topic	0.00	1.00	0.76	0.64	0.57
Tense	0.00	0.00	1.00	0.76	0.64
Negation	0.00	0.00	0.00	1.00	0.76
Sentiment	0.00	0.00	0.00	0.00	−1.00
LSTM · cell c_t					
Topic	0.00	0.88	0.83	0.80	0.76
Tense	0.00	0.00	0.64	0.57	0.50
Negation	0.00	0.00	0.00	0.95	0.04
Sentiment	0.00	0.00	0.00	0.00	−0.84

TOKEN: 1 · the | 2 · movie | 3 · was | 4 · not | 5 · good

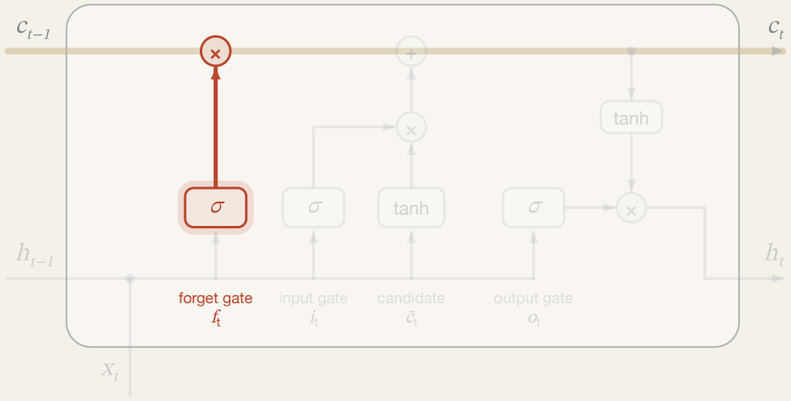
12

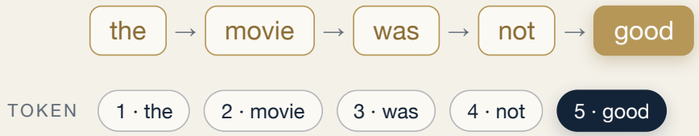
Forget gate: what to drop.

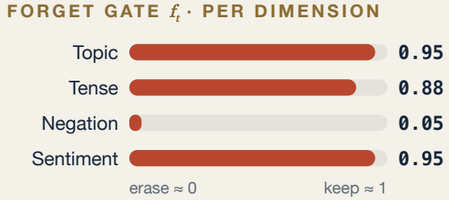
For each dimension, the forget gate outputs a number between **0 and 1**: keep or erase. It multiplies c_{t-1}.

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

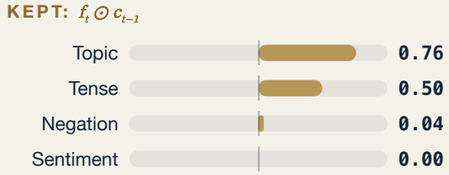
$$kept = f_t \odot c_{t-1}$$











Hand-set toy LSTM with four interpretable units (dummy word vectors, no training). At “good” the negation unit’s gate drops to **0.05**: the negation has done its job (it flipped “good”) and is erased. Everything else is kept (0.88–0.95).

13

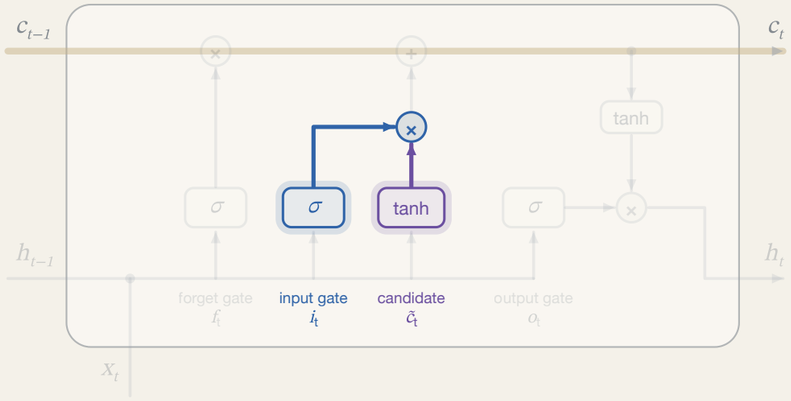
Input gate + candidate: what new info to write.

The **candidate** proposes what could be written; the **input gate** decides how much of it to actually write.

$$\tilde{c}_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_c)$$

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

propose (tanh) × gate (σ)



the → movie → was → not → good

TOKEN | 1 · the | 2 · movie | 3 · was | 4 · not | 5 · good

CANDIDATE $\tilde{c}_t$ · WHAT COULD BE WRITTEN

Topic	0.00
Tense	0.00
Negation	1.00
Sentiment	0.00

INPUT GATE i_t · HOW MUCH

Topic	0.12
Tense	0.12
Negation	0.95
Sentiment	0.00

WRITTEN: $i_t \odot \tilde{c}_t$

Topic	0.00
Tense	0.00
Negation	0.95
Sentiment	0.00

At “not” the candidate proposes **negation = 0.995** and the gate opens to **0.95** for that unit only. At “good” the candidate for sentiment is **–0.89**: it reads the negation stored in h_{t-1}, so a positive word proposes a negative sentiment.

14

Updating memory — by addition.

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$$

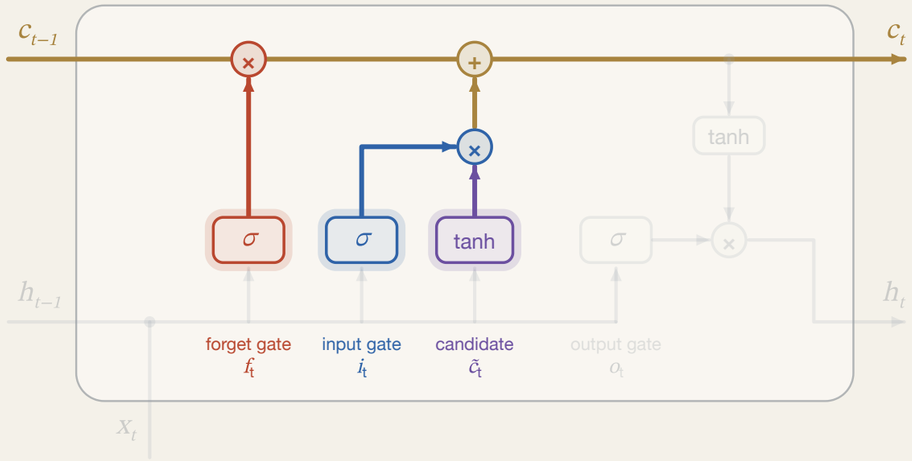
keep some old — add some new

	c_{t-1}	$\times f_t$	$= kept$	$+ i_t \odot \tilde{c}_t$	$= c_t$
Topic	0.83	0.95	0.80	0.00	0.80
Tense	0.64	0.88	0.57	0.00	0.57
Negation	0.00	0.99	0.00	0.95	0.95
Sentiment	0.00	0.95	0.00	0.00	0.00

the → movie → was → not → good

TOKEN: 1 · the | 2 · movie | 3 · was | 4 · not | 5 · good

Real values of the toy LSTM. At “not” the only big change is the negation unit (0.00 → 0.95); the topic and the tense are just multiplied by their forget gates. Nothing is pushed through a squashing function on the way.



The “+” is the whole trick.

15

Output gate: what to expose.

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

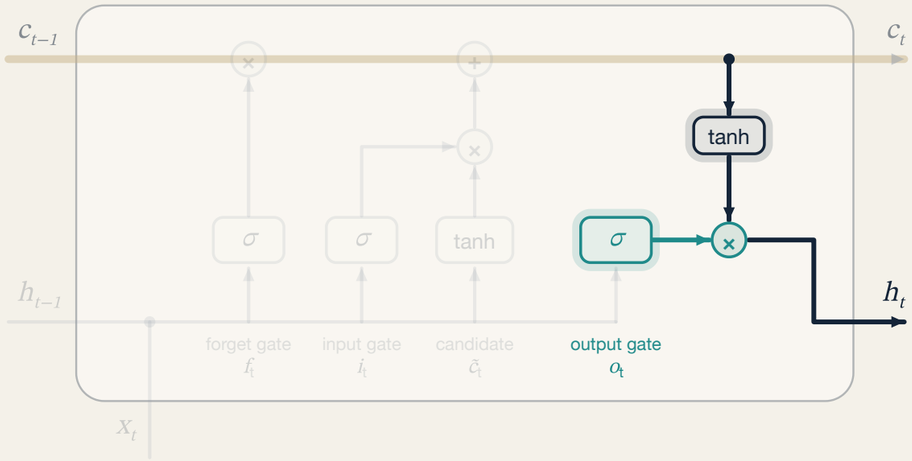
$$h_t = o_t \odot \tanh(c_t)$$

c_t is internal memory; **h_t** is the filtered view the rest of the network sees.

	c_t	$\tanh(c_t)$	$\times\, o_t$	$= h_t$
Topic	0.76	0.64	0.27	0.17
Tense	0.50	0.46	0.27	0.12
Negation	0.04	0.04	0.88	0.04
Sentiment	−0.84	−0.69	0.95	−0.66

the → movie → was → not → **good**

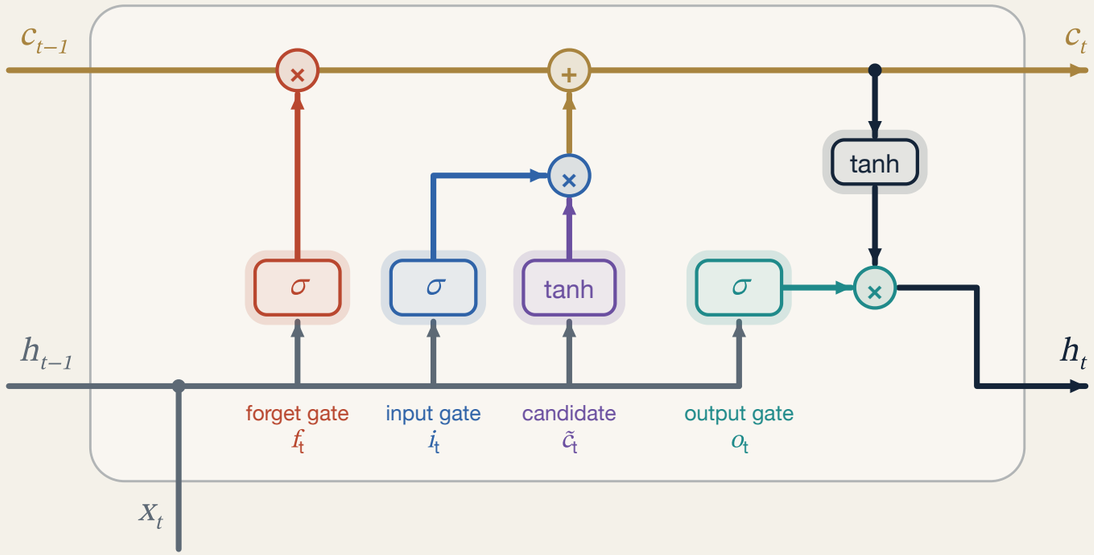
TOKEN 1 · the | 2 · movie | 3 · was | 4 · not | 5 · good



At “good”: the topic is still in memory (c = 0.76) but barely exposed (o = 0.27 → h = 0.17); the sentiment is exposed almost fully (o = 0.95 → h = −0.66), which is what the classifier reads.

16

The full cell, assembled.



- $f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$
  decides what to **forget**
- $i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$
  decides how much to **write**
- $\tilde{c}_t = \tanh(W_c \cdot [h_{t-1}, x_t] + b_c)$
  proposes new content
- $c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$
  updates the **memory highway**
- $o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$
  decides how much to **expose**
- $h_t = o_t \odot \tanh(c_t)$
  the **exposed** memory

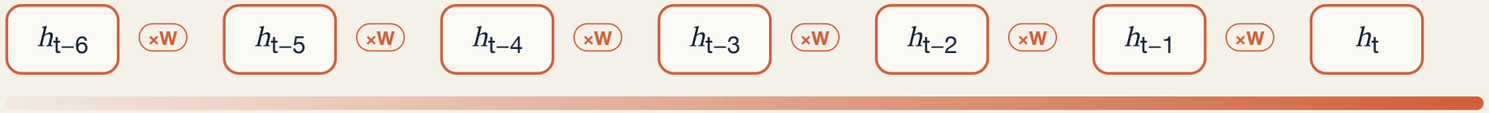
f i c̃ c o h all

17

Why it works: additive update = preserved signal.

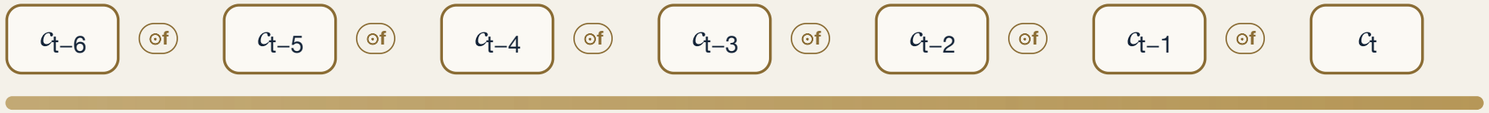
Plain RNN

The signal is **multiplied** by W and squashed by tanh at every step.



LSTM cell state

The signal flows **additively**: each step only multiplies by $f_t \approx 1$.



constant error carousel

HOW MUCH OF THE SIGNAL SURVIVES K STEPS



DISTANCE K

k = 5 | k = 10 | **k = 20** | k = 50 | k = 100

additive update = preserved signal.

18

GRU: fewer gates, often comparable.

LSTM — 3 gates (forget / input / output); separate c_t and h_t.

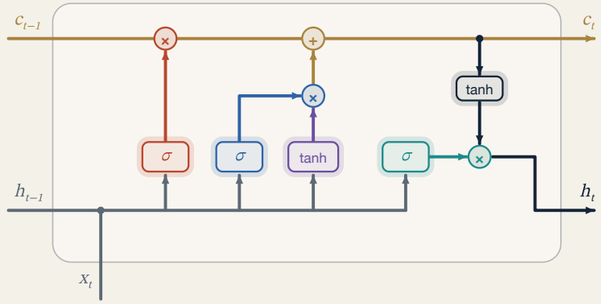
GRU — 2 gates (reset / update); a single merged state.

Frequently matches the LSTM at lower cost: the practitioner's **default to try**.

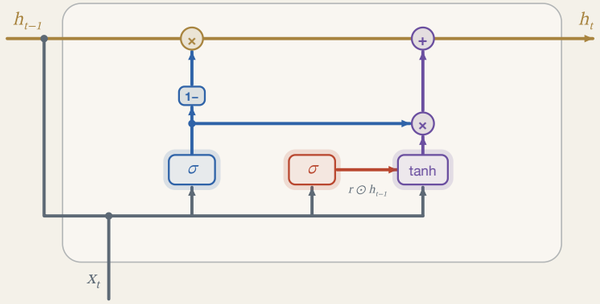
	LSTM	GRU
GATES	3 (forget, input, output)	2 (reset, update)
STATE LINES	2 (c_t and h_t)	1 (h)
RELATIVE COST	Higher	Lower
WEIGHTS · 300 → 128	**220,160**	**165,120** −25%

Real PyTorch counts for one layer with 300-d inputs and 128 units: the GRU has exactly **3/4** of the LSTM's weights (three weight blocks instead of four).

LSTM



GRU



19

GRU: the two gates, step by step.

$$z_t = \sigma(W_z \cdot [h_{t-1}, x_t] + b_z)$$

$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t] + b_r)$$

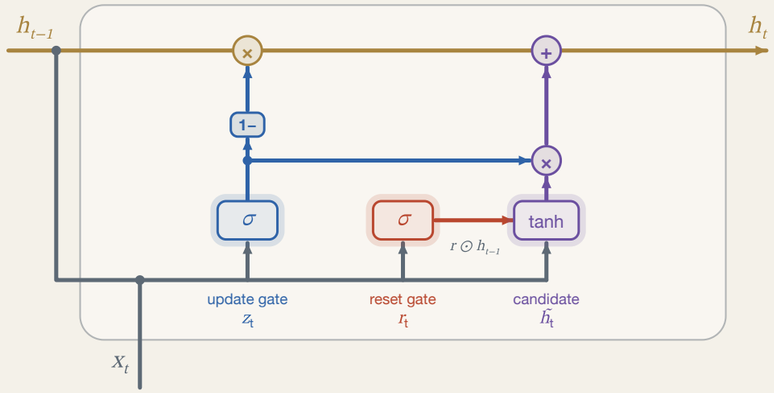
$$\tilde{h}_t = \tanh(W_h \cdot [r_t \odot h_{t-1}, x_t] + b_h)$$

$$h_t = (1 - z_t) \odot h_{t-1} + z_t \odot \tilde{h}_t$$

Update gate z: how much of each unit to rewrite. 1 – z keeps (the LSTM's forget gate), z writes (its input gate), coupled.

Reset gate r: how much of the old state the candidate may look at.

> ⚠ **PyTorch writes it the other way round**
>
> $h_t = (1 - z_t) \odot n_t + z_t \odot h_{t-1}$ (as Cho et al., 2014): the same model, but there z means "keep".



the → movie → was → not → good

TOKEN | 1 · the | 2 · movie | 3 · was | 4 · not | 5 · good

Toy GRU, hand-set weights. At "good" the update gate rewrites the negation unit with an empty candidate (erased) and writes the sentiment, read through the reset gate from the stored negation.

UPDATE GATE z_t · HOW MUCH TO REWRITE

Topic	0.01
Tense	0.05
Negation	0.98
Sentiment	0.95

RESET GATE r_t

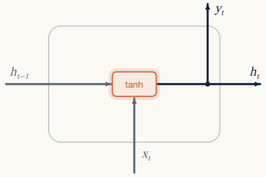
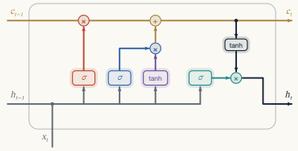
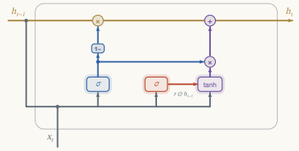
Topic	0.95
Tense	0.95
Negation	0.95
Sentiment	0.95

NEW STATE h_t

Topic	0.96
Tense	0.89
Negation	0.02
Sentiment	−0.95

20

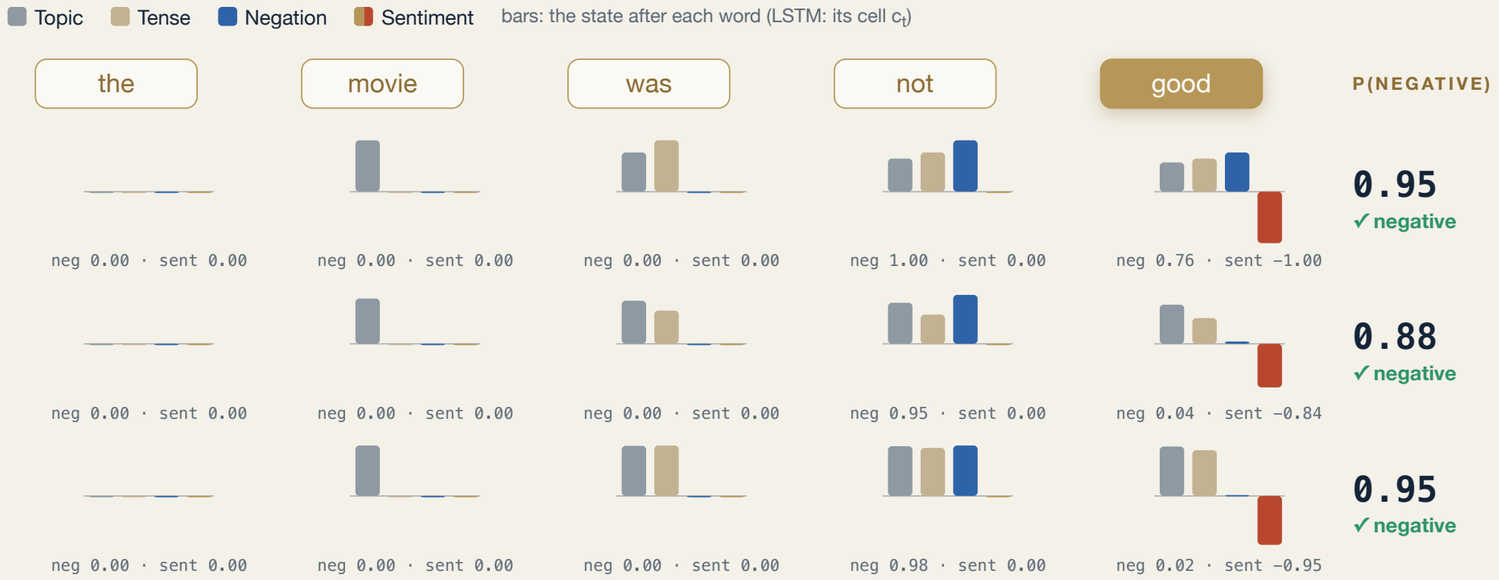
RNN vs LSTM vs GRU, side by side.

	Plain RNN 	LSTM 	GRU 
GATES	none	3 (f, i, o)	2 (z, r)
STATE	h_t	c_t and h_t	h_t
UPDATE	$h_t = \tanh(W \cdot [h_{t-1}, x_t] + b)$	$c_t = f \odot c_{t-1} + i \odot \tilde{c}$	$h_t = (1-z) \odot h_{t-1} + z \odot \tilde{h}$
WEIGHTS (PYTORCH)	$h(x+h) + 2h$	$4 \cdot [h(x+h) + 2h]$	$3 \cdot [h(x+h) + 2h]$
X = 300, H = 128	**55,040**	**220,160**	**165,120**
FORWARD, 32 × 100 STEPS	**2.7 ms**	**4.1 ms** (8.8 ms without its oneDNN kernel)	**6.3 ms**
TOY MEMORY OF “NOT”	lost “not” at a gap of **7** words	kept it up to a gap of **94**	kept it up to a gap of **120**

Real PyTorch 2.15.0 modules, one layer, forward only, 2-thread CPU, input 64, hidden 128 (median of 3 × 40 runs). The GRU does 3/4 of the LSTM’s arithmetic, but here only the LSTM runs on an optimized oneDNN kernel; with plain kernels the GRU is faster (5.9 ms vs 8.8 ms). Memory: the hand-set toy cells of the next two slides.

21

Example 1: one sentence, three cells.



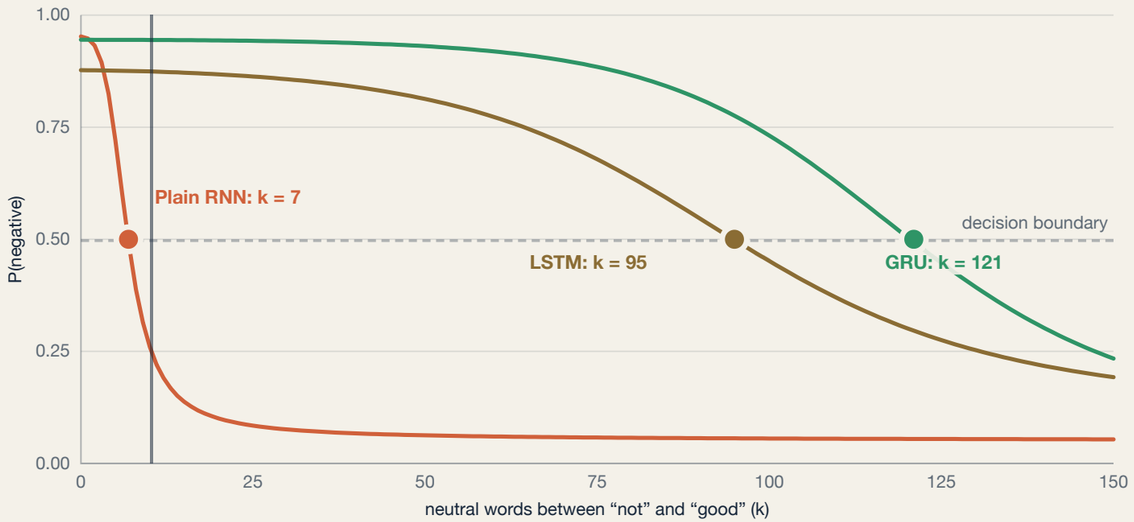
Control sentence “the movie was good”: P(positive) = **0.95** (RNN), **0.89** (LSTM), **0.94** (GRU). Same dummy word vectors, same readout for all three (P(positive) = σ(3 · sentiment unit)); hand-set weights, no training. With “not” right before “good”, **all three get it right**.

TOKEN 1 · the | 2 · movie | 3 · was | 4 · not | 5 · good

22

Example 2: remembering “not” across a gap.

the movie was **not** … **10** neutral words … **good**



AT THE SELECTED DISTANCE

Plain RNN
negation left 0.36
P(negative) **0.26**
✕ says positive

LSTM
negation left 0.63
P(negative) **0.87**
✓ negative

GRU
negation left 0.91
P(negative) **0.94**
✓ negative

K 0 5 **10** 50 100 150

Per neutral word the stored negation is multiplied by: **plain RNN** — re-squashed by tanh (×0.76 at first, then slowly); **LSTM** — its forget gate, f = 0.993; **GRU** — its keep factor, 1 – z = 0.993. The crossing points depend on these hand-set numbers; the shape (fast loss without gates, slow loss with them) does not.

23

When to use which.

Plain RNN

USE IT FOR

- very short dependencies
- tiny models and devices
- baselines and teaching

In our toy it lost the “not” at a gap of 7 words.

LSTM

USE IT FOR

- long dependencies, enough data
- when a separate memory helps
- the most studied variant

Seen in: speech recognition (Graves, Mohamed & Hinton, 2013); Google’s neural machine translation (Wu et al., 2016: 8 encoder + 8 decoder LSTM layers).

GRU

USE IT FOR

- fewer weights (3/4 of an LSTM)
- smaller datasets or memory budgets
- often comparable accuracy

Seen in: the encoder–decoder where it was introduced (Cho et al., 2014); music and speech modelling, on par with the LSTM (Chung et al., 2014).

Try both LSTM and GRU
no universal winner (Chung 2014; Jozefowicz 2015; Greff 2017)

Forget-gate bias = 1
for LSTMs (Jozefowicz 2015); PyTorch does not do it by default

Clip gradients
clip_grad_norm_ with max_norm around 1 to 5

Mask and pack
padded batches: next slides

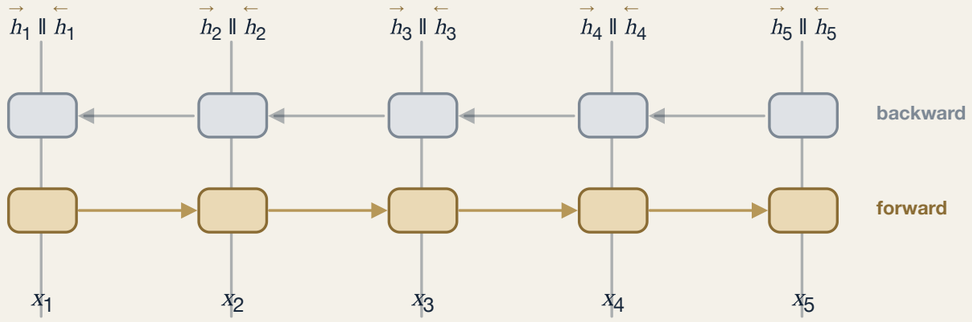
Very long context?
attention or state-space models

24

Two standard extensions.

Bidirectional

Read the sequence **forward and backward**, concatenate the outputs.



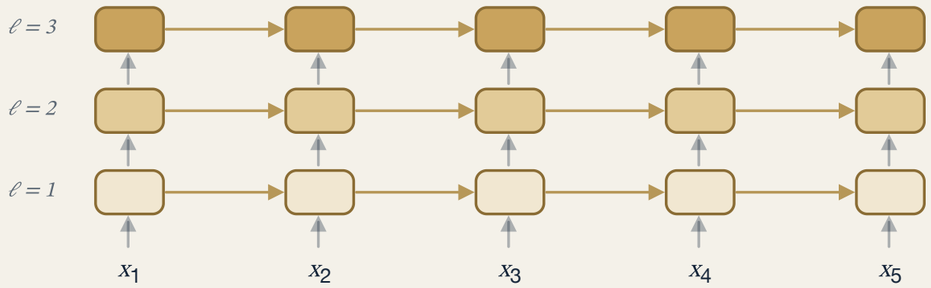
✕ **streaming**

Needs the full sequence: unusable in streaming or causal settings.

```
nn.LSTM(300, 128,
  bidirectional=True)
output (32, 20, 256)
440,320 weights (2 × 220,160)
```

Stacked / deep

The hidden states of layer ℓ **feed layer $\ell + 1$**.




```
nn.LSTM(300, 128, num_layers=2)
352,256 weights
+ bidirectional=True:
output (32, 20, 256)
h_n (4, 32, 128)
  = 2 layers × 2 directions
835,584 weights
```

25

Where people actually get stuck: padding & masking.

Pad sequences to equal length · **mask** padded positions so they don't pollute the loss · **pack** so the RNN skips the padding.

PADDED BATCH · MAX LENGTH T = 6

seq	length	t_1	t_2	t_3	t_4	t_5	t_6
1	3	t_1	t_2	t_3	PAD	PAD	PAD
2	5	t_1	t_2	t_3	t_4	t_5	PAD
3	2	t_1	t_2	PAD	PAD	PAD	PAD
4	6	t_1	t_2	t_3	t_4	t_5	t_6

MASK · 1 = REAL TOKEN, 0 = PAD

seq	length	t_1	t_2	t_3	t_4	t_5	t_6
1	3	1	1	1	0	0	0
2	5	1	1	1	1	1	0
3	2	1	1	0	0	0	0
4	6	1	1	1	1	1	1

PER-TOKEN LOSSES ℓ (DUMMY NUMBERS)

0.35	0.63	1.64	~~5.32~~	~~3.09~~	~~5.12~~
1.06	0.49	1.52	0.40	0.90	~~4.89~~
0.98	1.26	~~5.17~~	~~3.66~~	~~5.49~~	~~4.97~~
1.45	0.73	0.20	1.95	0.74	0.77

$$L = \frac{\Sigma(mask \odot \ell)}{\Sigma\, mask}$$

15.07 / 16 = **0.942**

masked positions contribute 0

LOSS ⚠ without mask | with mask

Efficiency tip: pack the batch

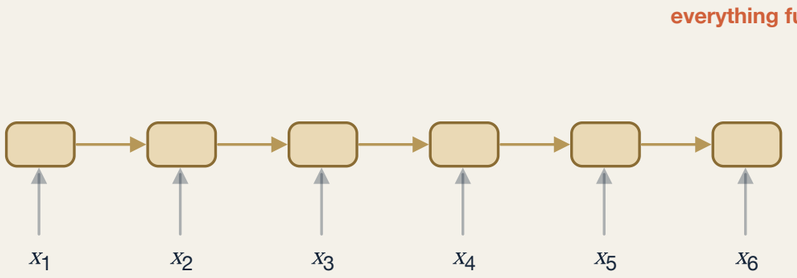
A random GRU (forward only) on this batch: if the padded batch is run to the end, the last state of each sequence keeps changing on the PAD steps. Difference from the packed run, ‖Δh‖: **0.55** · **0.39** · **0.60** · **0.00** for lengths 3, 5, 2, 6. Packing (or reading each output at its true last step) gives exactly the packed result.

```
pack_padded_sequence(x, lengths, batch_first=True, enforce_sorted=False)
```

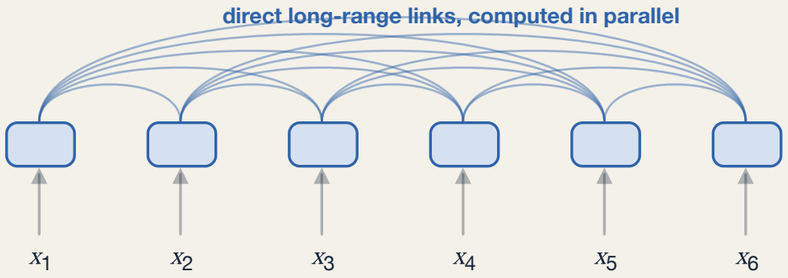
26

What replaced them — and why.

RNN



Transformer



1. Transformers **parallelize across the sequence**; RNNs are inherently sequential.

2. Attention models long-range dependencies **directly**, instead of squeezing them through one hidden state.

LAYER	WORK PER LAYER	SEQUENTIAL OPERATIONS	LONGEST PATH BETWEEN TWO WORDS
recurrent	$O(n \cdot d^2)$	**O(n)**	**O(n)**
self-attention	$O(n^2 \cdot d)$	**O(1)**	**O(1)**

n = sequence length, d = representation size (Vaswani et al., 2017, Table 1). Attention pays with $O(n^2)$ work, but that work runs in parallel; an RNN must take n steps one after another.

27

Recurrence isn't dead — and it's coming back.



Still used

- **streaming** and low-latency systems: one step per new input, constant memory
- **edge** and embedded devices
- many **time-series** tasks

Resurgent

- **state-space models** (Mamba) and **RWKV**: recurrence reengineered for parallel training
- **xLSTM** (2024): the LSTM with exponential gates and a matrix memory
- **minGRU / minLSTM** (2024): gates that depend only on x_t, so training parallelizes

streaming

edge

time-series

The vanishing-gradient problem you learned today is the same problem transformers solve differently: **that's next session**.

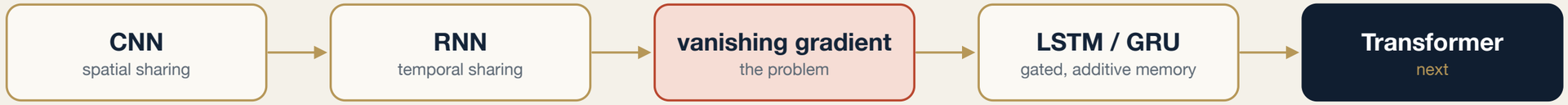
28

Five things people get wrong.

✕ “the output h_t is the memory”	✓ No — **h_t is filtered** by the output gate; **c_t is the real memory**. toy LSTM at “good”: topic c = 0.76, but h = 0.17
✕ “the hidden state remembers everything”	✓ No — it’s a **lossy summary**. 6 numbers summarize “the cat sat on the” (slide 4)
✕ “LSTM remembers everything”	✓ No — it **learns what to keep**. toy LSTM, f = 0.993: the “not” fades after ~95 words
✕ “deep RNN = long sequence”	✓ No — **depth in time** and **stacked layers** are different axes. num_layers stacks cells; the sequence length is T
✕ “a GRU is just a worse, cheaper LSTM”	✓ No — **3/4 of the weights**, and no clear winner across tasks. Chung et al., 2014 · Greff et al., 2017

29

The whole lecture in five lines.



THROUGH-LINE	weight sharing across time.
PROBLEM	the signal vanishes over distance.
FIX	an additive cell state + gates (LSTM), or a gated interpolation (GRU).
TAKEAWAY	**additive update = preserved signal.**
NEXT	how transformers solve the same problem differently.

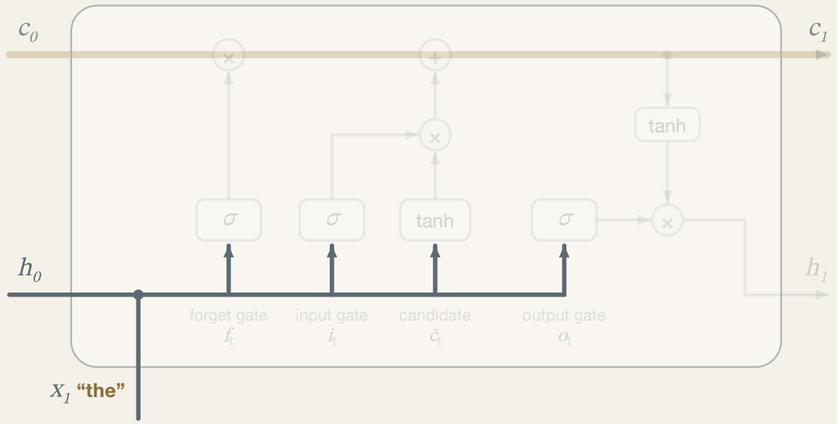
30

Worked example — t = 1: ‘the’.

Sentence: **the** → movie → was → not → good

HAND-SET TOY LSTM · EXACT VALUES, NOT TRAINED

- **Task:** classify the sentiment at the end of the sentence.
- At t = 1 the model sees a function word with little semantic content.
- So it writes very little into memory: every input gate is at most 0.12.



GATE VALUES · $t = 1$

	Topic	Tense	Neg.	Sent.
f_t	0.95	0.88	0.99	0.95
i_t	0.12	0.12	0.05	0.00
$\tilde{c}_t$	0.00	0.00	0.00	0.00
o_t	0.27	0.27	0.88	0.27

MEMORY c_1 AFTER THIS STEP

Topic	0.00
Tense	0.00
Negation	0.00
Sentiment	0.00

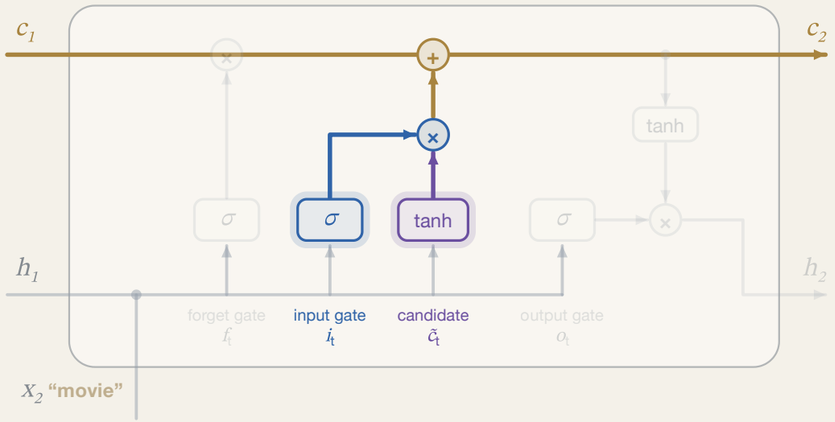
Interpretation: Mostly setup: almost no content worth storing yet. The memory stays at zero.

31

Worked example — t = 2: ‘movie’.

Sentence: the → **movie** → was → not → good

- **Task:** classify the sentiment at the end of the sentence.
- At t = 2 the token “movie” carries the first important semantic content.
- So the LSTM writes the topic strongly into memory: i = 0.88, c̃ = 0.99.



HAND-SET TOY LSTM · EXACT VALUES, NOT TRAINED

GATE VALUES · $t = 2$

	Topic	Tense	Neg.	Sent.
f_t	0.95	0.88	0.99	0.95
i_t	0.88	0.12	0.05	0.00
$\tilde{c}_t$	1.00	0.00	0.00	0.00
o_t	0.27	0.27	0.88	0.27

MEMORY c_2 AFTER THIS STEP

Topic	0.88
Tense	0.00
Negation	0.00
Sentiment	0.00

Interpretation: Now the model knows the sentence is about a movie (topic: c = 0.88). Still no sentiment.

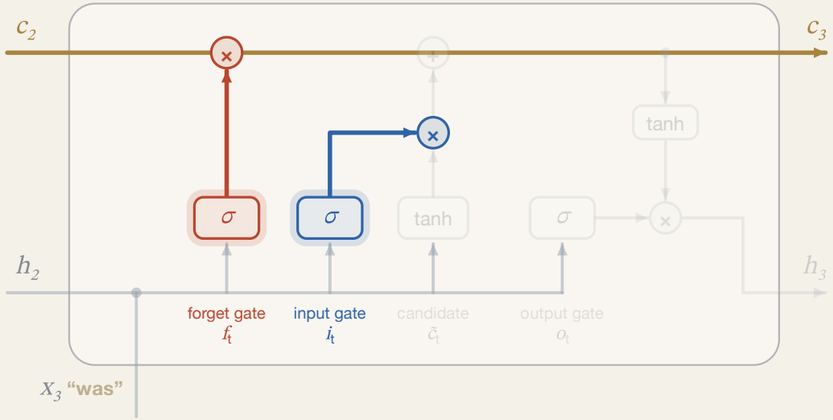
32

Worked example — t = 3: ‘was’.

Sentence: the → movie → **was** → not → good

HAND-SET TOY LSTM · EXACT VALUES, NOT TRAINED

- **Task:** classify the sentiment at the end of the sentence.
- At t = 3 “was” contributes grammar more than meaning.
- So the LSTM mostly keeps what it already knows (f = 0.95 for the topic), with a moderate update for tense.



GATE VALUES · $t = 3$

	Topic	Tense	Neg.	Sent.
f_t	0.95	0.88	0.99	0.95
i_t	0.12	0.65	0.05	0.00
$\tilde{c}_t$	0.00	1.00	0.00	0.00
o_t	0.27	0.27	0.88	0.27

MEMORY c_3 AFTER THIS STEP

Topic	0.83
Tense	0.64
Negation	0.00
Sentiment	0.00

Interpretation: Mostly preserve the topic (0.88 → 0.83) and add a grammatical cue (tense: 0.64) while waiting for more meaning.

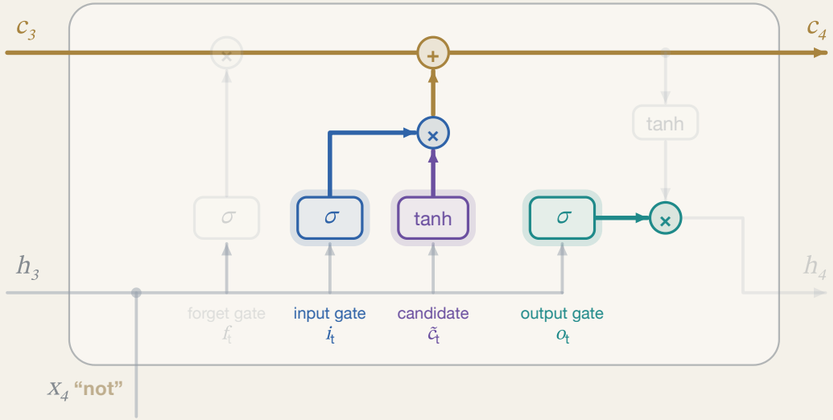
33

Worked example — t = 4: ‘not’.

Sentence: the → movie → was → **not** → good

HAND-SET TOY LSTM · EXACT VALUES, NOT TRAINED

- **Task:** classify the sentiment at the end of the sentence.
- At t = 4 “not” is the crucial token: it changes how a later adjective should be interpreted.
- So the LSTM writes negation strongly into memory (i = 0.95) and exposes it (o = 0.88).



GATE VALUES · $t = 4$

	Topic	Tense	Neg.	Sent.
f_t	0.95	0.88	0.99	0.95
i_t	0.12	0.12	0.95	0.00
$\tilde{c}_t$	0.00	0.00	1.00	0.00
o_t	0.27	0.27	0.88	0.27

MEMORY c_4 AFTER THIS STEP

Topic	0.80
Tense	0.57
Negation	0.95
Sentiment	0.00

Interpretation: The key memory step: negation is stored (c = 0.95) and visible in h_t (0.65), ready to modify the next opinion word.

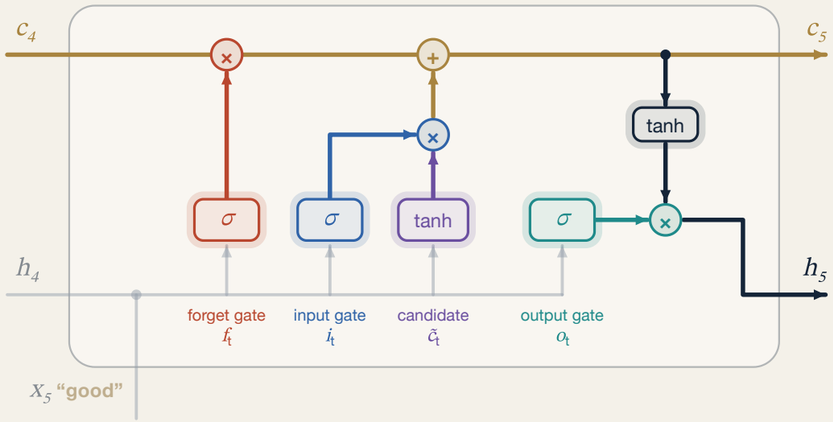
34

Worked example — t = 5: ‘good’.

Sentence: the → movie → was → not → good

HAND-SET TOY LSTM · EXACT VALUES, NOT TRAINED

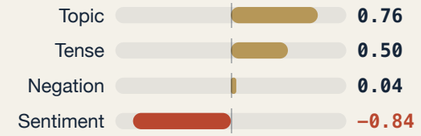
- **Task:** classify the sentiment at the end of the sentence.
- At t = 5 “good” is positive on its own, but the candidate also reads the stored negation.
- So the candidate proposes –0.89 for sentiment, the gate writes it (i = 0.95), and the used-up negation is erased (f = 0.05).



GATE VALUES · $t = 5$

	Topic	Tense	Neg.	Sent.
f_t	0.95	0.88	0.05	0.95
i_t	0.12	0.12	0.05	0.95
$\tilde{c}_t$	0.00	0.00	0.00	−0.89
o_t	0.27	0.27	0.88	0.95

MEMORY c_5 AFTER THIS STEP



Interpretation: The remembered negation flips “good” to negative: sentiment c = –0.84, exposed as h = –0.66, read by the classifier.

FINAL PREDICTION

Negative 0.88 · Positive 0.12

$P(\text{positive}) = \sigma(3 \cdot h_{\text{sentiment}}) = \sigma(3 \cdot -0.66)$