Sequence Models I: RNNs, BPTT, LSTMs and GRUs

Remembering with a hidden state — and gating what to keep

Prof. Nipun Batra

ES 667 · Deep Learning · IIT Gandhinagar

From fixed windows to recurrent state

Where we are

Last lecture built a language model over a **fixed window**: predict the next token from the previous k.

Where we are

Last lecture built a language model over a **fixed window**: predict the next token from the previous k.

$$p_\theta(x_{t+1} \mid x_{<t}) \approx p_\theta(x_{t+1} \mid x_{t-k+1:t})$$

Where we are

Last lecture built a language model over a **fixed window**: predict the next token from the previous k.

$$p_\theta(x_{t+1} \mid x_{<t}) \approx p_\theta(x_{t+1} \mid x_{t-k+1:t})$$

This lecture keeps the same **task** — next-token prediction — but changes **how the model remembers the past**: from a fixed window of inputs to a **running hidden state**.

Recall: modeling a sequence

D DERIVATION

A language model factorizes the joint probability by the chain rule:

Recall: modeling a sequence

D DERIVATION

A language model factorizes the joint probability by the chain rule:

$$p(x_{1:T}) = \prod_{t=1}^{T} p(x_t \mid x_{<t})$$

Recall: modeling a sequence

D DERIVATION

A language model factorizes the joint probability by the chain rule:

$$p(x_{1:T}) = \prod_{t=1}^{T} p(x_t \mid x_{<t})$$

and we train it by minimizing the negative log-likelihood — token-level cross-entropy:

$$\mathcal{L} = -\sum_{t=1}^{T} \log p_\theta(x_t \mid x_{<t})$$

Recall: modeling a sequence

D DERIVATION

A language model factorizes the joint probability by the chain rule:

$$p(x_{1:T}) = \prod_{t=1}^{T} p(x_t \mid x_{<t})$$

and we train it by minimizing the negative log-likelihood — token-level cross-entropy:

$$\mathcal{L} = -\sum_{t=1}^{T} \log p_\theta(x_t \mid x_{<t})$$

each factor is one V-way classification: “which token comes next?”

The fixed-window MLP, and its limits

The MLP-LM concatenates the last k embeddings and runs one MLP:

The fixed-window MLP, and its limits

The MLP-LM concatenates the last k embeddings and runs one MLP:

fixed context — anything before x_{t-k} is invisible; long-range dependencies are lost.

params grow with k — $W_h \in \mathbb{R}^{H \times kd}$; doubling context doubles the layer.

The fixed-window MLP, and its limits

The MLP-LM concatenates the last k embeddings and runs one MLP:

fixed context — anything before x_{t-k} is invisible; long-range dependencies are lost.

params grow with k — $W_h \in \mathbb{R}^{H \times kd}$; doubling context doubles the layer.

No shared state across time. The same token at position 1 vs 2 hits **different** columns of W_h — computation is not reused, and the input length is frozen at k.

What we actually want

A better sequence model should:

What we actually want

A better sequence model should:

- read **one token at a time**, for a sequence of **any length**;

What we actually want

A better sequence model should:

- read **one token at a time**, for a sequence of **any length**;
- keep a **memory** h_t — a summary of everything seen so far, $x_{1:t}$;

What we actually want

A better sequence model should:

- read **one token at a time**, for a sequence of **any length**;
- keep a **memory** h_t — a summary of everything seen so far, $x_{1:t}$;
- **predict** from that memory, $\hat{y}_t = g(h_t)$;

What we actually want

A better sequence model should:

- read **one token at a time**, for a sequence of **any length**;
- keep a **memory** h_t — a summary of everything seen so far, $x_{1:t}$;
- **predict** from that memory, $\hat{y}_t = g(h_t)$;
- **share** the same parameters at every step.

What we actually want

A better sequence model should:

- read **one token at a time**, for a sequence of **any length**;
- keep a **memory** h_t — a summary of everything seen so far, $x_{1:t}$;
- **predict** from that memory, $\hat{y}_t = g(h_t)$;
- **share** the same parameters at every step.

a running state updated by one function: $h_t = f_\theta(h_{t-1}, x_t)$

The recurrent idea

Replace the fixed window with a **state** that is updated at every step:

The recurrent idea

Replace the fixed window with a **state** that is updated at every step:

$$\underbrace{p(x_{t+1} \mid x_{t-k:t})}_{\text{MLP: fixed window}} \quad \longrightarrow \quad \underbrace{h_t = f_\theta(h_{t-1}, x_t)}_{\text{RNN: recurrent state}}$$

The recurrent idea

Replace the fixed window with a **state** that is updated at every step:

$$\underbrace{p(x_{t+1} \mid x_{t-k:t})}_{\text{MLP: fixed window}} \quad \longrightarrow \quad \underbrace{h_t = f_\theta(h_{t-1}, x_t)}_{\text{RNN: recurrent state}}$$

One **shared** function f_θ applied over and over gives us, for free:

The recurrent idea

Replace the fixed window with a **state** that is updated at every step:

$$\underbrace{p(x_{t+1} \mid x_{t-k:t})}_{\text{MLP: fixed window}} \quad \longrightarrow \quad \underbrace{h_t = f_\theta(h_{t-1}, x_t)}_{\text{RNN: recurrent state}}$$

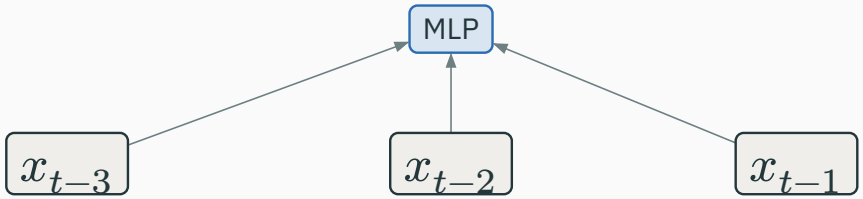
One **shared** function f_θ applied over and over gives us, for free:

Property	**Why it follows**
variable length	same f_θ runs for as many steps as there are tokens
parameter sharing	one f_θ, not one weight block per position
learned state	h_t is a trainable summary of $x_{1:t}$
online	update as each token arrives — no need to see the whole sequence

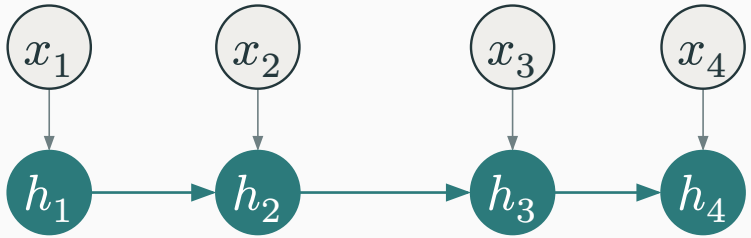
Fixed window vs recurrent state

V VISUAL

window $k = 3$ only



fixed window (MLP-LM)

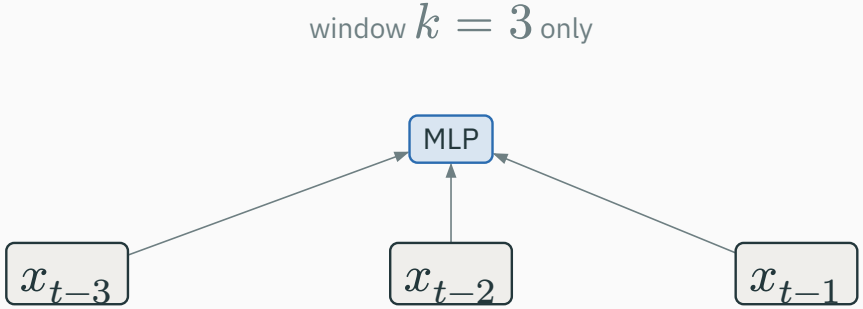


state carries all of the past

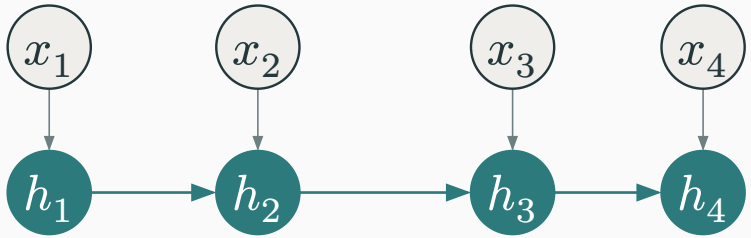
recurrent (RNN)

Fixed window vs recurrent state

V VISUAL



fixed window (MLP-LM)



state carries all of the past

recurrent (RNN)

the MLP forgets past the window; the RNN **carries** the past forward in h_t

The vanilla (Elman) RNN

The equations

D DERIVATION

At each step t: look up the token, update the state, read out a prediction.

The equations

D DERIVATION

At each step t: look up the token, update the state, read out a prediction.

$e_t = E[x_t]$ (embedding lookup)

The equations

D DERIVATION

At each step t: look up the token, update the state, read out a prediction.

$$e_t = E[x_t] \qquad \text{(embedding lookup)}$$

$$a_t = W_{xh}\, e_t + W_{hh}\, h_{t-1} + b_h, \qquad h_t = \varphi(a_t)$$

The equations

D DERIVATION

At each step t: look up the token, update the state, read out a prediction.

$$e_t = E[x_t] \qquad \text{(embedding lookup)}$$

$$a_t = W_{xh}\, e_t + W_{hh}\, h_{t-1} + b_h, \qquad h_t = \varphi(a_t)$$

$$z_t = W_{hy}\, h_t + b_y, \qquad p_t = \mathrm{softmax}(z_t)$$

The equations

D DERIVATION

At each step t: look up the token, update the state, read out a prediction.

$$e_t = E[x_t] \qquad \text{(embedding lookup)}$$

$$a_t = W_{xh}\, e_t + W_{hh}\, h_{t-1} + b_h, \qquad h_t = \varphi(a_t)$$

$$z_t = W_{hy}\, h_t + b_y, \qquad p_t = \mathrm{softmax}(z_t)$$

$\varphi = \tanh$ usually; one set of weights W_{xh}, W_{hh}, W_{hy} reused at **every** step (Elman)

The RNN cell

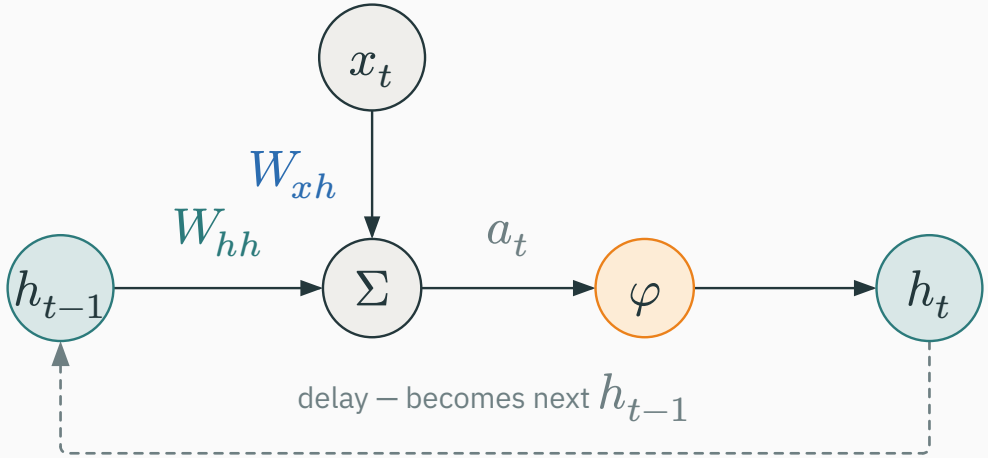
V VISUAL

One shared cell: the past state h_{t-1} and the new input x_t meet, then φ gives h_t.

The RNN cell

V VISUAL

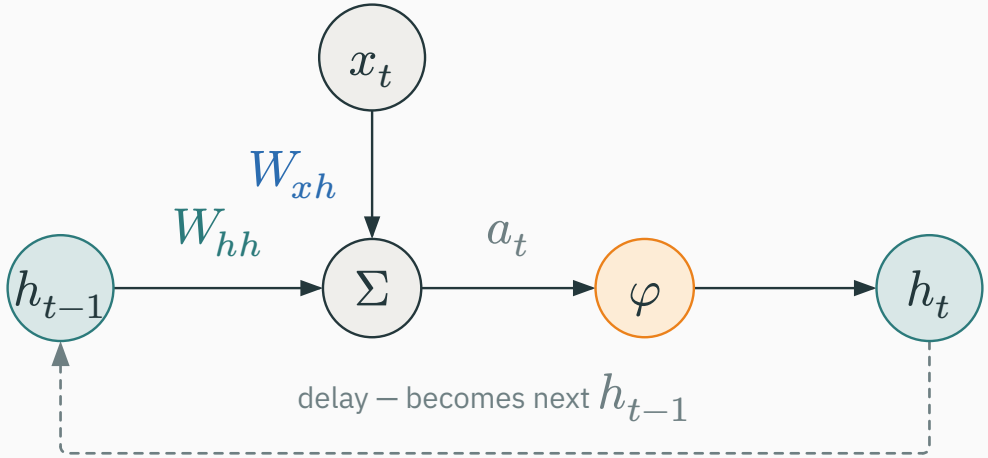
One shared cell: the past state h_{t-1} and the new input x_t meet, then φ gives h_t.



The RNN cell

V VISUAL

One shared cell: the past state h_{t-1} and the new input x_t meet, then φ gives h_t.



$h_t = \varphi(W_{xh} e_t + W_{hh} h_{t-1} + b_h)$ — the dashed edge is the **recurrence**

Unrolling the RNN in time

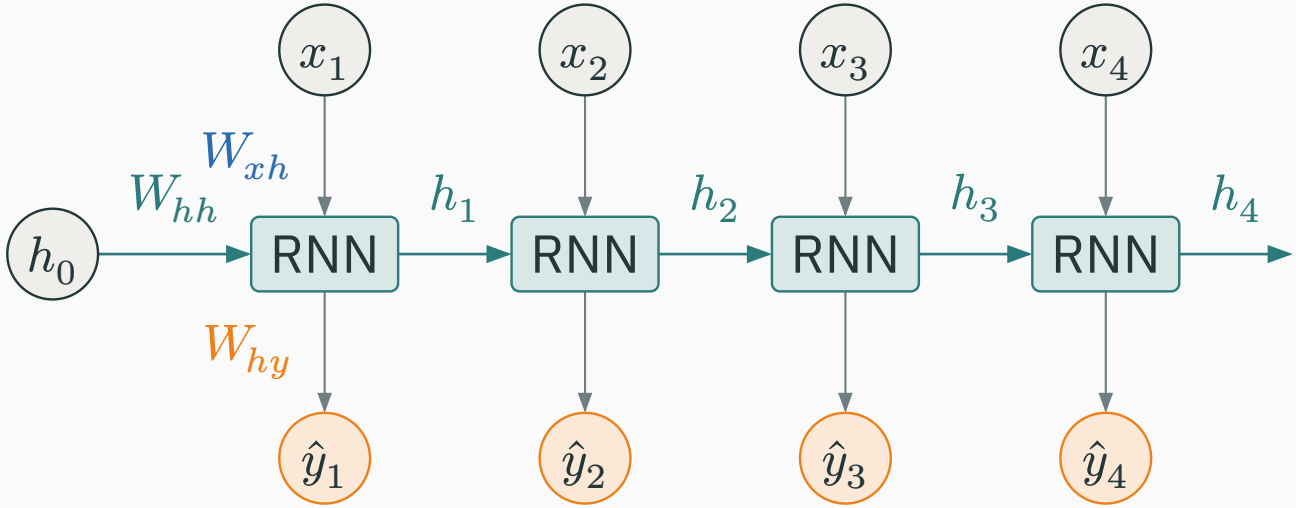
V VISUAL

The **same** cell, drawn once per step — this is what backprop actually sees:

Unrolling the RNN in time

V VISUAL

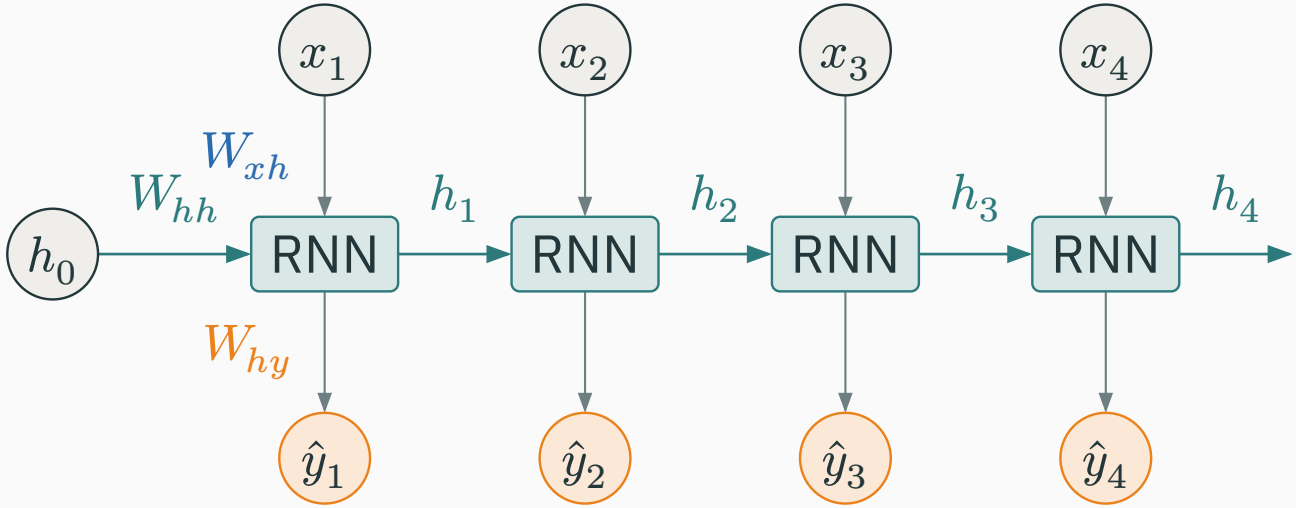
The **same** cell, drawn once per step — this is what backprop actually sees:



Unrolling the RNN in time

V VISUAL

The **same** cell, drawn once per step — this is what backprop actually sees:



W_{xh}, W_{hh}, W_{hy} are **shared** across all steps — the diagram repeats one cell

The hidden state is memory

D DERIVATION

Unroll the recurrence and the whole past is nested inside h_t:

The hidden state is memory

D DERIVATION

Unroll the recurrence and the whole past is nested inside h_t:

$$h_t = f(h_{t-1}, x_t) = f(f(h_{t-2}, x_{t-1}), x_t) = \ldots$$

The hidden state is memory

D DERIVATION

Unroll the recurrence and the whole past is nested inside h_t:

$$h_t = f(h_{t-1}, x_t) = f(f(h_{t-2}, x_{t-1}), x_t) = ...$$

$$h_t = f(f(...f(h_0, x_1)..., x_{t-1}), x_t)$$

The hidden state is memory

D DERIVATION

Unroll the recurrence and the whole past is nested inside h_t:

$$h_t = f(h_{t-1}, x_t) = f(f(h_{t-2}, x_{t-1}), x_t) = \dots$$

$$h_t = f(f(\dots f(h_0, x_1) \dots, x_{t-1}), x_t)$$

h_t is a **learned, fixed-size summary** of the entire prefix $x_{1:t}$ — no matter how long. The window is gone; the state remembers.

RNN for language modeling

D DERIVATION

Feed a sentence in; the **target at each step is the next token** (teacher forcing):

RNN for language modeling

D DERIVATION

Feed a sentence in; the **target at each step is the next token** (teacher forcing):

step t	1	2	3	4
input x_t	`deep`	`learning`	`is`	`fun`
state	h_1	h_2	h_3	h_4
target x_{t+1}	`learning`	`is`	`fun`	`<eos>`

RNN for language modeling

D DERIVATION

Feed a sentence in; the **target at each step is the next token** (teacher forcing):

step t	1	2	3	4
input x_t	`deep`	`learning`	`is`	`fun`
state	h_1	h_2	h_3	h_4
target x_{t+1}	`learning`	`is`	`fun`	`<eos>`

loss $= \sum_t$ cross-entropy(p_t, x_{t+1}) — every position supervises the step before it

RNN task patterns

V VISUAL

The same cell, wired differently, covers most sequence tasks:

RNN task patterns

V VISUAL

The same cell, wired differently, covers most sequence tasks:

Pattern	Shape	Example
many-to-one	$x_{1:T} \to y$	sentiment: read all, predict at h_T
many-to-many (aligned)	$x_{1:T} \to y_{1:T}$	POS tag each token
language model	$x_{1:T} \to x_{2:T+1}$	next-token at every step
seq2seq (encoder–decoder)	$x_{1:T} \to y_{1:U}$	translation — different length

Tensor shapes

Track a batch of B sequences, length T, embedding d, hidden m, vocab V:

Tensor shapes

Track a batch of B sequences, length T, embedding d, hidden m, vocab V:

Tensor	Shape	Meaning
token ids X	$B \times T$	one id per position
embeddings $E[X]$	$B \times T \times d$	one d-vector per token
hidden states H	$B \times T \times m$	the state at every step
logits Z	$B \times T \times V$	a next-token score at every step

Tensor shapes

Track a batch of B sequences, length T, embedding d, hidden m, vocab V:

Tensor	Shape	Meaning
token ids X	$B \times T$	one id per position
embeddings $E[X]$	$B \times T \times d$	one d-vector per token
hidden states H	$B \times T \times m$	the state at every step
logits Z	$B \times T \times V$	a next-token score at every step

time is now an **axis**, not a fixed slot — the model handles any T

A fully-worked RNN forward pass

The tiny scalar RNN

Strip it to **one** hidden unit — everything is a scalar so we can do it by hand:

The tiny scalar RNN

Strip it to **one** hidden unit — everything is a scalar so we can do it by hand:

$$h_t = \tanh(w_x\, x_t + w_h\, h_{t-1} + b)$$

The tiny scalar RNN

Strip it to **one** hidden unit — everything is a scalar so we can do it by hand:

$$h_t = \tanh(w_x\, x_t + w_h\, h_{t-1} + b)$$

$$w_x = 1, \quad w_h = 0.5, \quad b = 0, \quad h_0 = 0, \quad x = (1, 2, -1)$$

The tiny scalar RNN

Strip it to **one** hidden unit — everything is a scalar so we can do it by hand:

$$h_t = \tanh(w_x\, x_t + w_h\, h_{t-1} + b)$$

$$w_x = 1, \quad w_h = 0.5, \quad b = 0, \quad h_0 = 0, \quad x = (1, 2, -1)$$

three steps, three states — watch the memory build

Step 1

D DERIVATION

Plug $x_1 = 1$ and $h_0 = 0$ into the update:

Step 1

D DERIVATION

Plug $x_1 = 1$ and $h_0 = 0$ into the update:

$$a_1 = w_x x_1 + w_h h_0 + b$$

Step 1

D DERIVATION

Plug $x_1 = 1$ and $h_0 = 0$ into the update:

$$a_1 = w_x x_1 + w_h h_0 + b$$

$$a_1 = 1 \cdot 1 + 0.5 \cdot 0 + 0 = 1$$

Step 1

D DERIVATION

Plug $x_1 = 1$ and $h_0 = 0$ into the update:

$$a_1 = w_x x_1 + w_h h_0 + b$$

$$a_1 = 1 \cdot 1 + 0.5 \cdot 0 + 0 = 1$$

$$h_1 = \tanh(1) \approx 0.762$$

Step 1

D DERIVATION

Plug $x_1 = 1$ and $h_0 = 0$ into the update:

$$a_1 = w_x x_1 + w_h h_0 + b$$

$$a_1 = 1 \cdot 1 + 0.5 \cdot 0 + 0 = 1$$

$$h_1 = \tanh(1) \approx 0.762$$

with no past state ($h_0 = 0$), the first step is just the input through $\tanh$

Step 2

D DERIVATION

Now the past enters through $w_h h_1$:

Step 2

D DERIVATION

Now the past enters through $w_h h_1$:

$$a_2 = w_x x_2 + w_h h_1 = 1 \cdot 2 + 0.5 \cdot 0.762$$

Step 2

D DERIVATION

Now the past enters through $w_h h_1$:

$$a_2 = w_x x_2 + w_h h_1 = 1 \cdot 2 + 0.5 \cdot 0.762$$

$$a_2 = 2 + 0.381 = 2.381$$

Step 2

D DERIVATION

Now the past enters through $w_h h_1$:

$$a_2 = w_x x_2 + w_h h_1 = 1 \cdot 2 + 0.5 \cdot 0.762$$

$$a_2 = 2 + 0.381 = 2.381$$

$$h_2 = \tanh(2.381) \approx 0.983$$

Step 2

D DERIVATION

Now the past enters through $w_h h_1$:

$$a_2 = w_x x_2 + w_h h_1 = 1 \cdot 2 + 0.5 \cdot 0.762$$

$$a_2 = 2 + 0.381 = 2.381$$

$$h_2 = \tanh(2.381) \approx 0.983$$

the state is being pushed toward the $+1$ saturation of $\tanh$

Step 3

D DERIVATION

A negative input, but the state still remembers the positive past:

Step 3

D DERIVATION

A negative input, but the state still remembers the positive past:

$$a_3 = w_x x_3 + w_h h_2 = 1 \cdot (-1) + 0.5 \cdot 0.983$$

Step 3

D DERIVATION

A negative input, but the state still remembers the positive past:

$$a_3 = w_x x_3 + w_h h_2 = 1 \cdot (-1) + 0.5 \cdot 0.983$$

$$a_3 = -1 + 0.492 = -0.509$$

Step 3

D DERIVATION

A negative input, but the state still remembers the positive past:

$$a_3 = w_x x_3 + w_h h_2 = 1 \cdot (-1) + 0.5 \cdot 0.983$$

$$a_3 = -1 + 0.492 = -0.509$$

$$h_3 = \tanh(-0.509) \approx -0.469$$

Step 3

D DERIVATION

A negative input, but the state still remembers the positive past:

$$a_3 = w_x x_3 + w_h h_2 = 1 \cdot (-1) + 0.5 \cdot 0.983$$

$$a_3 = -1 + 0.492 = -0.509$$

$$h_3 = \tanh(-0.509) \approx -0.469$$

the $+0.49$ carried from h_2 **softened** the -1 input — that is memory at work

The forward trace

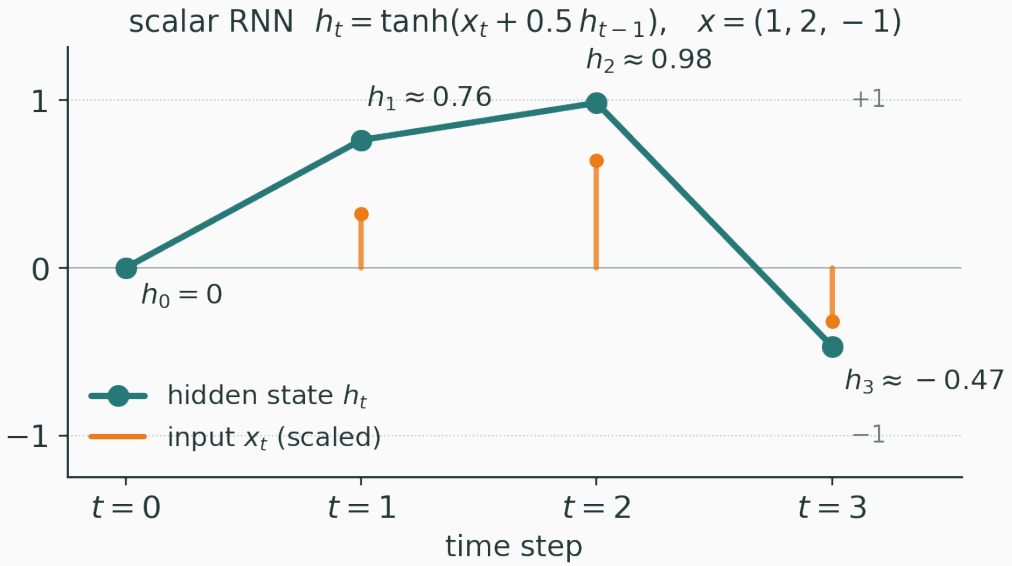
V VISUAL

t	x_t	$a_t = x_t + 0.5h_{t-1}$	$h_t = \tanh(a_t)$
1	$+1$	1.000	0.762
2	$+2$	2.381	0.983
3	-1	-0.509	-0.469

The forward trace

V VISUAL

t	x_t	$a_t = x_t + 0.5h_{t-1}$	$h_t = \tanh(a_t)$
1	$+1$	1.000	0.762
2	$+2$	2.381	0.983
3	-1	-0.509	-0.469



Why this is memory

D DERIVATION

Substitute the recurrence backwards:

Why this is memory

D DERIVATION

Substitute the recurrence backwards:

$$h_3 = \tanh\left(x_3 + 0.5 \underbrace{\tanh\left(x_2 + 0.5 \underbrace{\tanh(x_1)}_{h_1} \right)}_{h_2} \right)$$

Why this is memory

D DERIVATION

Substitute the recurrence backwards:

$$h_3 = \tanh\Bigg(x_3 + 0.5 \tanh\underbrace{\Bigg(x_2 + 0.5 \underbrace{\tanh(x_1)}_{h_1} \Bigg)}_{h_2} \Bigg)$$

h_3 nests **every** earlier input — each one $\tanh$ deeper, with one more factor of w_h

Why this is memory (cont.)

D DERIVATION

$$h_3 = \tanh\left(x_3 + 0.5 \underbrace{\tanh\left(x_2 + 0.5 \underbrace{\tanh(x_1)}_{h_1} \right)}_{h_2} \right)$$

Why this is memory (cont.)

D DERIVATION

$$h_3 = \tanh\Bigg(x_3 + 0.5 \tanh\Bigg(\underbrace{x_2 + 0.5 \underbrace{\tanh(x_1)}_{h_1}}_{h_2} \Bigg) \Bigg)$$

x_1 reaches h_3 through **two** nested $\tanh$ and two factors of $w_h = 0.5$. Information persists — but each step **attenuates** it by w_h. Hold that thought (Part V).

Interactive: RNN forward

I INTERACTIVE

INTERACTIVE

↗ nipunbatra.github.io/interactive-articles/rnn-state-stepper

set w_x, w_h, b and type a short sequence; watch h_t update step by step and the state trajectory trace out, exactly like the scalar example above.

Backpropagation through time

An RNN is a deep net in time

D DERIVATION

Unrolled, the RNN is just a **very deep** feedforward net whose layers **share weights**:

An RNN is a deep net in time

D DERIVATION

Unrolled, the RNN is just a **very deep** feedforward net whose layers **share weights**:

$h_0 \to h_1 \to h_2 \to \ldots \to h_T$

An RNN is a deep net in time

D DERIVATION

Unrolled, the RNN is just a **very deep** feedforward net whose layers **share weights**:

$h_0 \to h_1 \to h_2 \to \ldots \to h_T$

training it = backprop on the unrolled graph = backpropagation through time (BPTT)

An RNN is a deep net in time

D DERIVATION

Unrolled, the RNN is just a **very deep** feedforward net whose layers **share weights**:

$$h_0 \to h_1 \to h_2 \to \ldots \to h_T$$

training it = backprop on the unrolled graph = backpropagation through time (BPTT)

same chain rule as Lecture 3 — only now the "depth" is the sequence length T

The loss sums over time

D DERIVATION

Every step contributes a term; the total loss is their sum:

The loss sums over time

D DERIVATION

Every step contributes a term; the total loss is their sum:

$$\mathcal{L} = \sum_{t=1}^{T} \mathcal{L}_t, \qquad \mathcal{L}_t = -\log p_\theta(x_{t+1} \mid x_{\le t})$$

The loss sums over time

D DERIVATION

Every step contributes a term; the total loss is their sum:

$$\mathcal{L} = \sum_{t=1}^{T} \mathcal{L}_t, \qquad \mathcal{L}_t = -\log p_\theta(x_{t+1} \mid x_{\le t})$$

Because the weights are **shared**, each step's gradient lands on the **same** parameters:

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

The loss sums over time

D DERIVATION

Every step contributes a term; the total loss is their sum:

$$\mathcal{L} = \sum_{t=1}^{T} \mathcal{L}_t, \qquad \mathcal{L}_t = -\log p_\theta(x_{t+1} \mid x_{\le t})$$

Because the weights are **shared**, each step's gradient lands on the **same** parameters:

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

this is exactly Lecture 3's rule: a shared parameter **accumulates** gradient from every use

The backward recurrence

D DERIVATION

Write $\overline{u} = \partial \mathcal{L} / \partial u$. Walking one step back through the cell:

The backward recurrence

D DERIVATION

Write $\overline{u} = \partial \mathcal{L} / \partial u$. Walking one step back through the cell:

$$\overline{a}_t = \overline{h}_t \odot \varphi'(a_t) \qquad \text{(through the tanh)}$$

The backward recurrence

D DERIVATION

Write $\overline{u} = \partial \mathcal{L} / \partial u$. Walking one step back through the cell:

$$\overline{a}_t = \overline{h}_t \odot \varphi'(a_t) \qquad \text{(through the tanh)}$$

$$\nabla_{W_{xh}} \mathrel{+}= \overline{a}_t \, e_t^\top, \qquad \nabla_{W_{hh}} \mathrel{+}= \overline{a}_t \, h_{t-1}^\top$$

The backward recurrence

D DERIVATION

Write $\overline{u} = \partial \mathcal{L} / \partial u$. Walking one step back through the cell:

$$\overline{a}_t = \overline{h}_t \odot \varphi'(a_t) \qquad \text{(through the tanh)}$$

$$\nabla_{W_{xh}} \mathrel{+}= \overline{a}_t \, e_t^\top, \qquad \nabla_{W_{hh}} \mathrel{+}= \overline{a}_t \, h_{t-1}^\top$$

$$\overline{h}_{t-1} \mathrel{+}= W_{hh}^\top \, \overline{a}_t \qquad \text{(state gradient to the previous step)}$$

Two sources of $\overline{h}_t$

D DERIVATION

The state h_t is used **twice** — so its gradient is a **sum** of two paths:

Two sources of $\overline{h}_t$

D DERIVATION

The state h_t is used **twice** — so its gradient is a **sum** of two paths:

$$\overline{h}_t = \underbrace{\frac{\partial \mathcal{L}_t}{\partial h_t}}_{\text{output path: its own loss}} + \underbrace{\frac{\partial \mathcal{L}_{\geq t+1}}{\partial h_t}}_{\text{future path: W\_(h h)\^top overline(a)\_(t+1)}}$$

Two sources of $\overline{h}_t$

D DERIVATION

The state h_t is used **twice** — so its gradient is a **sum** of two paths:

$$\overline{h}_t = \underbrace{\frac{\partial \mathcal{L}_t}{\partial h_t}}_{\text{output path: its own loss}} + \underbrace{\frac{\partial \mathcal{L}_{\geq t+1}}{\partial h_t}}_{\text{future path: W\_(h h)\^top overline(a)\_(t+1)}}$$

Exactly the **branch rule**: h_t feeds both this step's readout **and** the next step's state, so gradients from both add. Miss the second term and long-range learning is impossible.

BPTT, unrolled

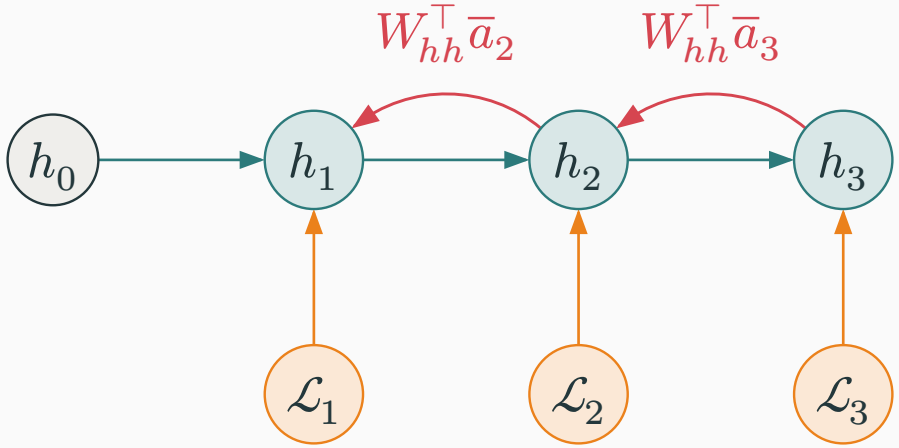
V VISUAL

Forward state (teal) flows right; the adjoint (orange = own loss, red = future) flows **back**:

BPTT, unrolled

V VISUAL

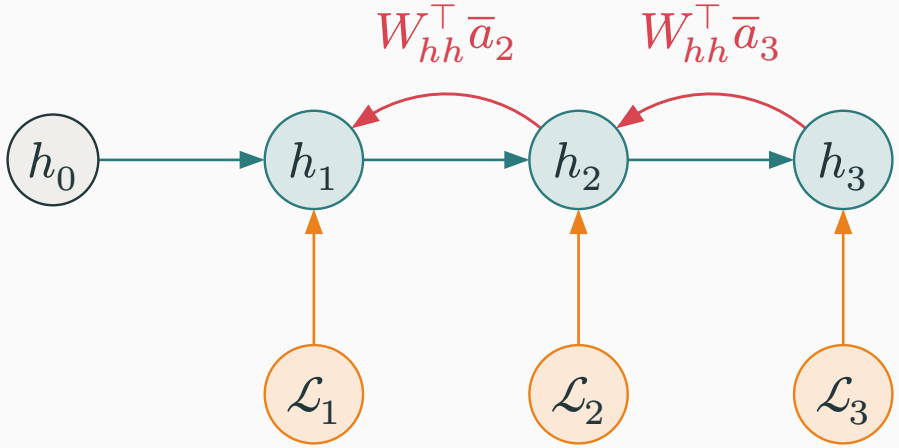
Forward state (teal) flows right; the adjoint (orange = own loss, red = future) flows **back**:



BPTT, unrolled

V VISUAL

Forward state (teal) flows right; the adjoint (orange = own loss, red = future) flows **back**:



each $\bar{h}_t$ collects its own loss (↓) plus the future via $W_{hh}^\top$ (←)

Truncated BPTT

D DERIVATION

Full BPTT over a length-10^4 sequence is expensive and unstable. So **chunk** it:

Truncated BPTT

D DERIVATION

Full BPTT over a length-10^4 sequence is expensive and unstable. So **chunk** it:

- process the sequence in blocks of length K (say $K = 35$);

Truncated BPTT

D DERIVATION

Full BPTT over a length-10^4 sequence is expensive and unstable. So **chunk** it:

- process the sequence in blocks of length K (say $K = 35$);
- backprop **within** a block; then **carry the state but detach it**:

Truncated BPTT

D DERIVATION

Full BPTT over a length-10^4 sequence is expensive and unstable. So **chunk** it:

- process the sequence in blocks of length K (say $K = 35$);
- backprop **within** a block; then **carry the state but detach it**:

$$h_{\text{start}}^{(\text{next block})} = \text{detach}\left(h_{\text{end}}^{(\text{this block})}\right)$$

Truncated BPTT

D DERIVATION

Full BPTT over a length-10^4 sequence is expensive and unstable. So **chunk** it:

- process the sequence in blocks of length K (say $K = 35$);
- backprop **within** a block; then **carry the state but detach it**:

$$h_{\text{start}}^{(\text{next block})} = \text{detach}\left(h_{\text{end}}^{(\text{this block})}\right)$$

the state keeps flowing forward; the **gradient** stops at the block boundary — bounded cost

Interactive: BPTT

I INTERACTIVE

INTERACTIVE

↗ nipunbatra.github.io/interactive-articles/bptt-unroll

step through the unrolled graph and watch each $\overline{h}_t$ accumulate its two contributions; toggle truncation length K and see where the gradient is cut.

Vanishing and exploding gradients

The long-range gradient is a product

D DERIVATION

To learn a dependency from step k to step T, the gradient runs the whole chain:

The long-range gradient is a product

D DERIVATION

To learn a dependency from step k to step T, the gradient runs the whole chain:

$$\frac{\partial h_T}{\partial h_k} = \prod_{j=k+1}^{T} \frac{\partial h_j}{\partial h_{j-1}}, \qquad \frac{\partial h_j}{\partial h_{j-1}} = D_j \, W_{hh}$$

The long-range gradient is a product

D DERIVATION

To learn a dependency from step k to step T, the gradient runs the whole chain:

$$\frac{\partial h_T}{\partial h_k} = \prod_{j=k+1}^{T} \frac{\partial h_j}{\partial h_{j-1}}, \qquad \frac{\partial h_j}{\partial h_{j-1}} = D_j \, W_{hh}$$

where $D_j = \operatorname{diag}\big(\varphi'\big(a_j\big)\big)$ is the activation slope at step j.

The long-range gradient is a product

D DERIVATION

To learn a dependency from step k to step T, the gradient runs the whole chain:

$$\frac{\partial h_T}{\partial h_k} = \prod_{j=k+1}^{T} \frac{\partial h_j}{\partial h_{j-1}}, \qquad \frac{\partial h_j}{\partial h_{j-1}} = D_j \, W_{hh}$$

where $D_j = \operatorname{diag}\big(\varphi'\big(a_j\big)\big)$ is the activation slope at step j.

a product of $(T-k)$ matrices — it tends to vanish or explode

The scalar recurrence lays it bare

D DERIVATION

Take one unit, $h_t = \tanh(w\, h_{t-1})$. Near 0, $\tanh'(0) = 1$, so:

The scalar recurrence lays it bare

D DERIVATION

Take one unit, $h_t = \tanh(w\, h_{t-1})$. Near 0, $\tanh'(0) = 1$, so:

$$\frac{\partial h_T}{\partial h_0} \approx w^T$$

The scalar recurrence lays it bare

D DERIVATION

Take one unit, $h_t = \tanh(w\, h_{t-1})$. Near 0, $\tanh'(0) = 1$, so:

$$\frac{\partial h_T}{\partial h_0} \approx w^T$$

$|w| < 1$: $w^T \to 0$ — **vanishing**. The distant past cannot send a gradient.

$|w| > 1$: $w^T \to \infty$ — **exploding**. Gradients blow up, training diverges.

Worked: w^{50}

D DERIVATION

Fifty steps back, with two nearby weights. Read powers as $w^{50} = e^{50 \ln w}$:

Worked: w^{50}

D DERIVATION

Fifty steps back, with two nearby weights. Read powers as $w^{50} = e^{50 \ln w}$:

$$w = 0.9: \quad 50 \ln 0.9 \approx 50 \cdot (-0.105) = -5.27$$

Worked: w^{50}

D DERIVATION

Fifty steps back, with two nearby weights. Read powers as $w^{50} = e^{50 \ln w}$:

$$w = 0.9: \quad 50 \ln 0.9 \approx 50 \cdot (-0.105) = -5.27$$

$$0.9^{50} = e^{-5.27} \approx 0.005 \quad \text{(vanished)}$$

Worked: w^{50}

D DERIVATION

Fifty steps back, with two nearby weights. Read powers as $w^{50} = e^{50 \ln w}$:

$$w = 0.9: \quad 50 \ln 0.9 \approx 50 \cdot (-0.105) = -5.27$$

$$0.9^{50} = e^{-5.27} \approx 0.005 \quad \text{(vanished)}$$

$$w = 1.1: \quad 50 \ln 1.1 \approx 50 \cdot 0.0953 = 4.77$$

Worked: w^{50}

D DERIVATION

Fifty steps back, with two nearby weights. Read powers as $w^{50} = e^{50 \ln w}$:

$$w = 0.9: \quad 50 \ln 0.9 \approx 50 \cdot (-0.105) = -5.27$$

$$0.9^{50} = e^{-5.27} \approx 0.005 \quad \text{(vanished)}$$

$$w = 1.1: \quad 50 \ln 1.1 \approx 50 \cdot 0.0953 = 4.77$$

$$1.1^{50} = e^{4.77} \approx 117 \quad \text{(exploded)}$$

Worked: w^{50}

D DERIVATION

Fifty steps back, with two nearby weights. Read powers as $w^{50} = e^{50 \ln w}$:

$$w = 0.9: \quad 50 \ln 0.9 \approx 50 \cdot (-0.105) = -5.27$$

$$0.9^{50} = e^{-5.27} \approx 0.005 \quad \text{(vanished)}$$

$$w = 1.1: \quad 50 \ln 1.1 \approx 50 \cdot 0.0953 = 4.77$$

$$1.1^{50} = e^{4.77} \approx 117 \quad \text{(exploded)}$$

a 10% change in w flips the gradient from $\frac{1}{200}$ to $\times\, 117$ — over just 50 steps

The saturation problem too

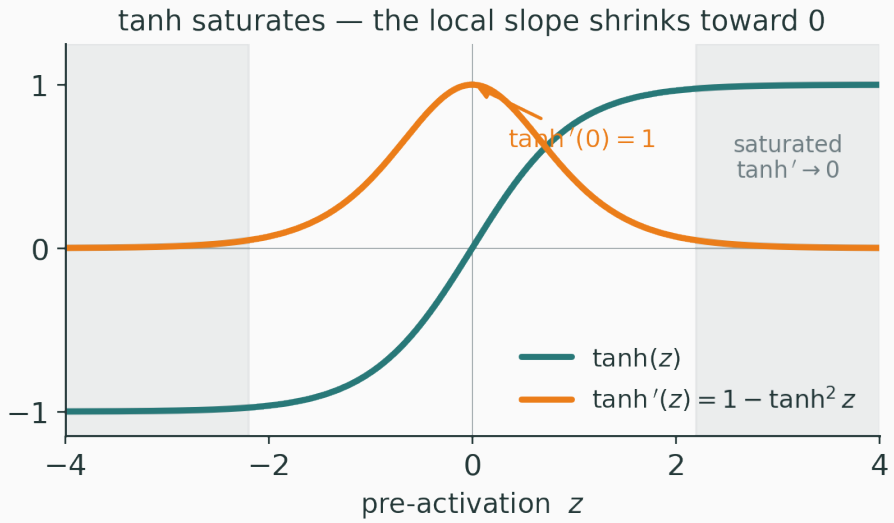
V VISUAL

$\tanh'(z) = 1 - \tanh^2(z) \leq 1$, and $\to 0$ once the unit saturates:

The saturation problem too

V VISUAL

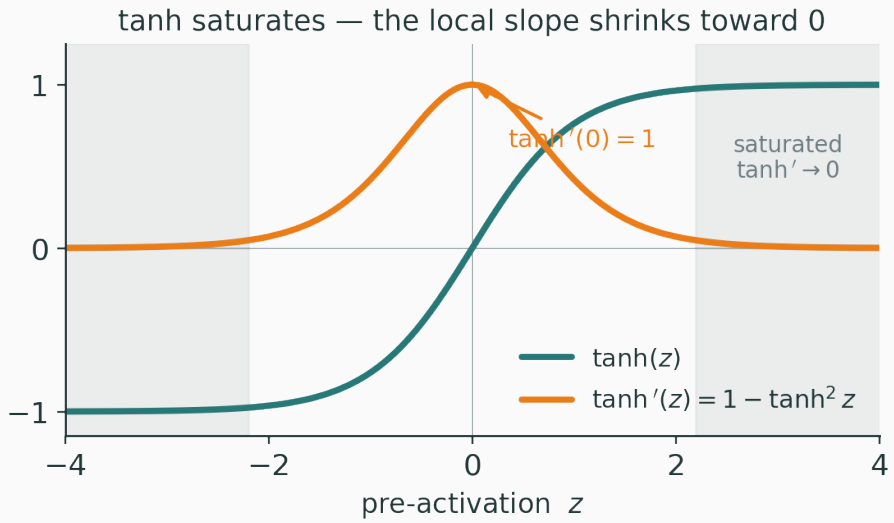
$\tanh'(z) = 1 - \tanh^2(z) \leq 1$, and $\to 0$ once the unit saturates:



The saturation problem too

V VISUAL

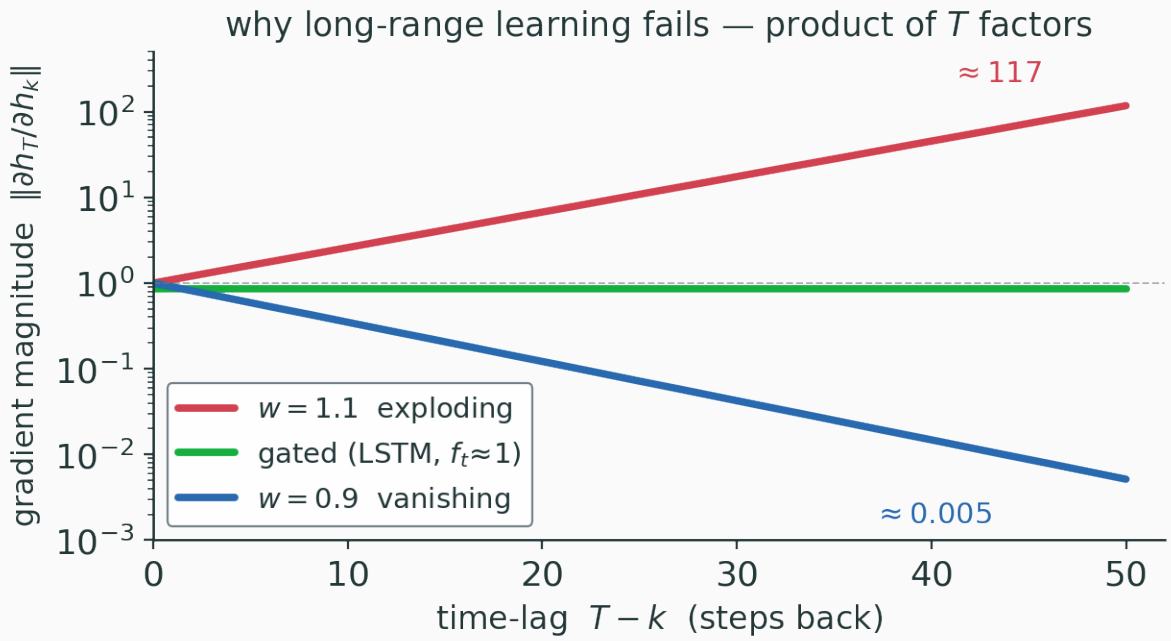
$\tanh'(z) = 1 - \tanh^2(z) \leq 1$, and $\to 0$ once the unit saturates:



each $D_j \leq 1$ can only **shrink** the signal further — saturation makes vanishing worse

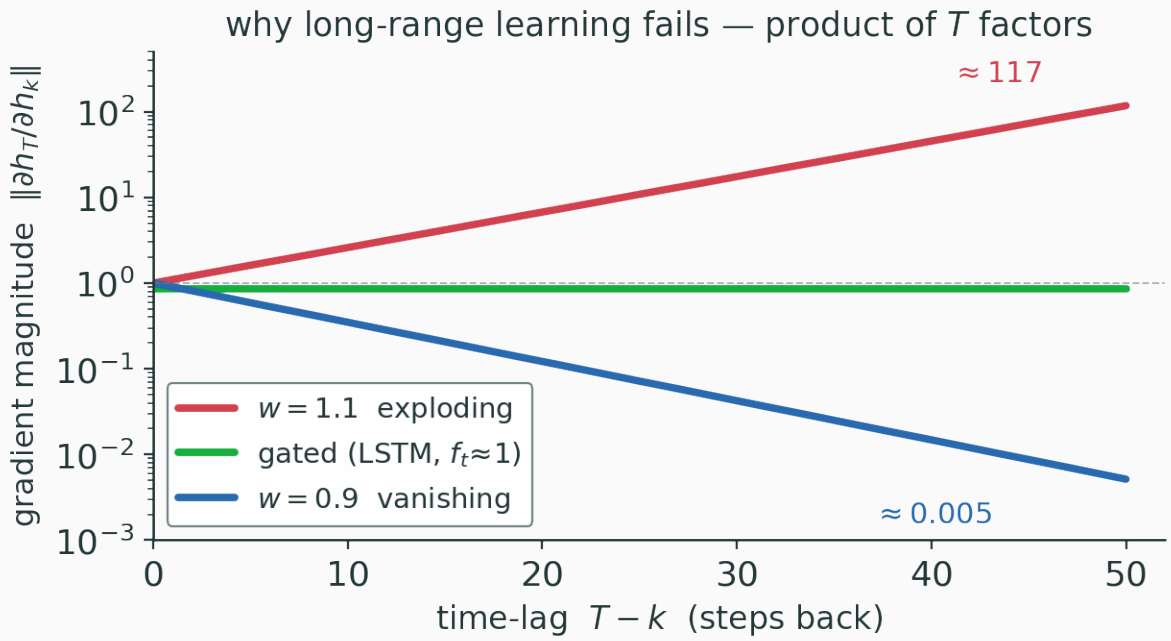
Gradient norm vs time-lag

V VISUAL



Gradient norm vs time-lag

V VISUAL



vanish (decay), explode (blow up), or — with a **gate** holding $f_t \approx 1$ — stay **flat**

Gradient clipping

D DERIVATION

Explosion has a cheap fix: cap the global gradient **norm** before the update.

Gradient clipping

D DERIVATION

Explosion has a cheap fix: cap the global gradient **norm** before the update.

$$\text{if } \|g\| > c: \quad g \leftarrow c\,\frac{g}{\|g\|}$$

Gradient clipping

D DERIVATION

Explosion has a cheap fix: cap the global gradient **norm** before the update.

$$\text{if } \|g\| > c: \quad g \leftarrow c\,\frac{g}{\|g\|}$$

Rescale, don't truncate — the **direction** of g is preserved, only its **length** is capped at c (Pascanu, Mikolov & Bengio 2013).

Worked: clipping

D DERIVATION

Threshold $c = 5$, gradient $g = (6, 8)$:

Worked: clipping

D DERIVATION

Threshold $c = 5$, gradient $g = (6, 8)$:

$$\|g\| = \sqrt{6^2 + 8^2} = \sqrt{36 + 64} = \sqrt{100} = 10$$

Worked: clipping

D DERIVATION

Threshold $c = 5$, gradient $g = (6, 8)$:

$$\|g\| = \sqrt{6^2 + 8^2} = \sqrt{36 + 64} = \sqrt{100} = 10$$

$$10 > c = 5 \quad \Rightarrow \quad \text{rescale by} \quad \frac{c}{\|g\|} = \frac{5}{10} = 0.5$$

Worked: clipping

D DERIVATION

Threshold $c = 5$, gradient $g = (6, 8)$:

$$\|g\| = \sqrt{6^2 + 8^2} = \sqrt{36 + 64} = \sqrt{100} = 10$$

$$10 > c = 5 \quad \Rightarrow \quad \text{rescale by} \quad \frac{c}{\|g\|} = \frac{5}{10} = 0.5$$

$$g \leftarrow 0.5 \cdot (6, 8) = (3, 4)$$

Worked: clipping

D DERIVATION

Threshold $c = 5$, gradient $g = (6, 8)$:

$$\|g\| = \sqrt{6^2 + 8^2} = \sqrt{36 + 64} = \sqrt{100} = 10$$

$$10 > c = 5 \quad \Rightarrow \quad \text{rescale by} \quad \frac{c}{\|g\|} = \frac{5}{10} = 0.5$$

$$g \leftarrow 0.5 \cdot (6, 8) = (3, 4)$$

$$\|(3, 4)\| = \sqrt{9 + 16} = 5 \quad \checkmark$$

Worked: clipping

D DERIVATION

Threshold $c = 5$, gradient $g = (6, 8)$:

$$\|g\| = \sqrt{6^2 + 8^2} = \sqrt{36 + 64} = \sqrt{100} = 10$$

$$10 > c = 5 \quad \Rightarrow \quad \text{rescale by} \quad \frac{c}{\|g\|} = \frac{5}{10} = 0.5$$

$$g \leftarrow 0.5 \cdot (6, 8) = (3, 4)$$

$$\|(3, 4)\| = \sqrt{9 + 16} = 5 \quad \checkmark$$

same direction $(3, 4) \parallel (6, 8)$, length now exactly 5

Clipping is only half the cure

Clipping is only half the cure

Fixes exploding — caps the step size so a single huge gradient cannot blow up training.

Does nothing for vanishing — you cannot rescale a gradient that has already decayed to ≈ 0.

Clipping is only half the cure

Fixes exploding — caps the step size so a single huge gradient cannot blow up training.

Does nothing for vanishing — you cannot rescale a gradient that has already decayed to ≈ 0.

to carry gradients across long lags we must change the architecture → gates

Interactive: gradient dynamics

I INTERACTIVE

INTERACTIVE

↗ nipunbatra.github.io/interactive-articles/vanishing-gradients

Sweep the recurrent weight w and the lag T; watch the long-range gradient collapse for $|w| < 1$, blow up for $|w| > 1$, and stay flat once a gate holds the memory open.

The LSTM

The core problem

D DERIVATION

The vanilla RNN **overwrites** its state every step: $h_t = \varphi(W_{hh} h_{t-1} + \ldots)$.

The core problem

D DERIVATION

The vanilla RNN **overwrites** its state every step: $h_t = \varphi(W_{hh} h_{t-1} + \ldots)$.

Every step multiplies the old state by W_{hh} and squashes it — memory is **rewritten**, and (Part V) the gradient is **multiplied** away. We want to **choose** what to keep.

The core problem

D DERIVATION

The vanilla RNN **overwrites** its state every step: $h_t = \varphi(W_{hh} h_{t-1} + \ldots)$.

Every step multiplies the old state by W_{hh} and squashes it — memory is **rewritten**, and (Part V) the gradient is **multiplied** away. We want to **choose** what to keep.

let the network decide, per component: what to forget, what to write, what to expose

Two states, one additive highway

D DERIVATION

The LSTM (Hochreiter & Schmidhuber, 1997) carries **two** vectors:

Two states, one additive highway

D DERIVATION

The LSTM (Hochreiter & Schmidhuber, 1997) carries **two** vectors:

cell state c_t — the long-term memory (the “highway”).

hidden state h_t — the exposed output, a filtered view of c_t.

Two states, one additive highway

D DERIVATION

The LSTM (Hochreiter & Schmidhuber, 1997) carries **two** vectors:

cell state c_t — the long-term memory (the "highway").

hidden state h_t — the exposed output, a filtered view of c_t.

The cell updates by **addition**, not full overwrite:

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

Two states, one additive highway

D DERIVATION

The LSTM (Hochreiter & Schmidhuber, 1997) carries **two** vectors:

cell state c_t — the long-term memory (the "highway").

hidden state h_t — the exposed output, a filtered view of c_t.

The cell updates by **addition**, not full overwrite:

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

keep a gated fraction of the old memory, **add** a gated bit of new — this is the whole idea

Gates are learned $[0, 1]$ valves

D DERIVATION

A **gate** is a vector in $[0, 1]$, computed by a sigmoid from the input and previous state:

Gates are learned $[0, 1]$ valves

D DERIVATION

A **gate** is a vector in $[0, 1]$, computed by a sigmoid from the input and previous state:

$$\text{gate} = \sigma(W\, u_t + b) \in (0, 1)^m, \qquad u_t = [h_{t-1}\ ;\ x_t]$$

Gates are learned $[0, 1]$ valves

D DERIVATION

A **gate** is a vector in $[0, 1]$, computed by a sigmoid from the input and previous state:

$$\text{gate} = \sigma(W\, u_t + b) \in (0, 1)^m, \qquad u_t = [h_{t-1}\ ;\ x_t]$$

0 = block the channel, 1 = let it pass — applied **elementwise** with $\odot$

The LSTM equations

D DERIVATION

Shared input $u_t = [h_{t-1}; x_t]$; three sigmoid gates and one $\tanh$ candidate:

The LSTM equations

D DERIVATION

Shared input $u_t = [h_{t-1}; x_t]$; three sigmoid gates and one $\tanh$ candidate:

$$f_t = \sigma\big(W_f u_t + b_f\big), \quad i_t = \sigma(W_i u_t + b_i), \quad o_t = \sigma(W_o u_t + b_o)$$

The LSTM equations

D DERIVATION

Shared input $u_t = [h_{t-1}; x_t]$; three sigmoid gates and one $\tanh$ candidate:

$$f_t = \sigma\big(W_f u_t + b_f\big), \quad i_t = \sigma(W_i u_t + b_i), \quad o_t = \sigma(W_o u_t + b_o)$$

$$\tilde{c}_t = \tanh(W_c u_t + b_c) \qquad \text{(candidate memory)}$$

The LSTM equations

D DERIVATION

Shared input $u_t = [h_{t-1}; x_t]$; three sigmoid gates and one $\tanh$ candidate:

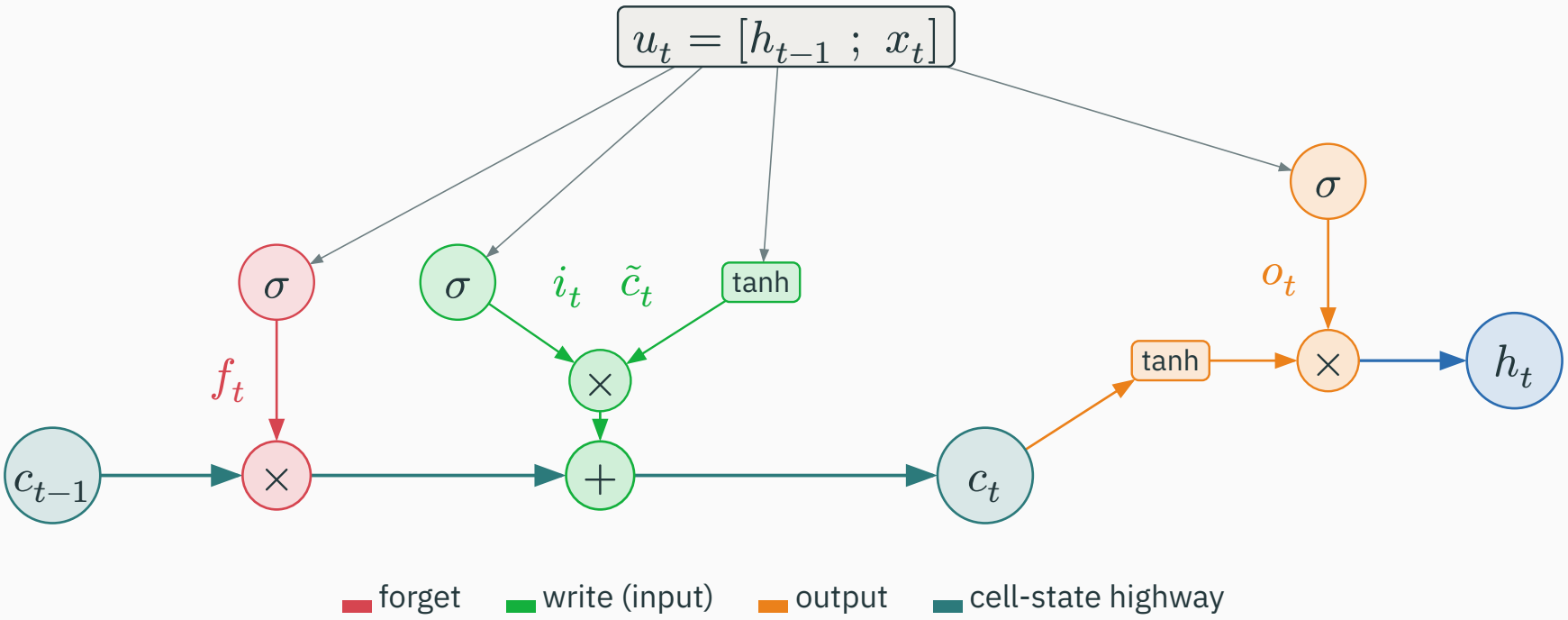
$$f_t = \sigma\big(W_f u_t + b_f\big), \quad i_t = \sigma(W_i u_t + b_i), \quad o_t = \sigma(W_o u_t + b_o)$$

$$\tilde{c}_t = \tanh(W_c u_t + b_c) \qquad \text{(candidate memory)}$$

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t, \qquad h_t = o_t \odot \tanh(c_t)$$

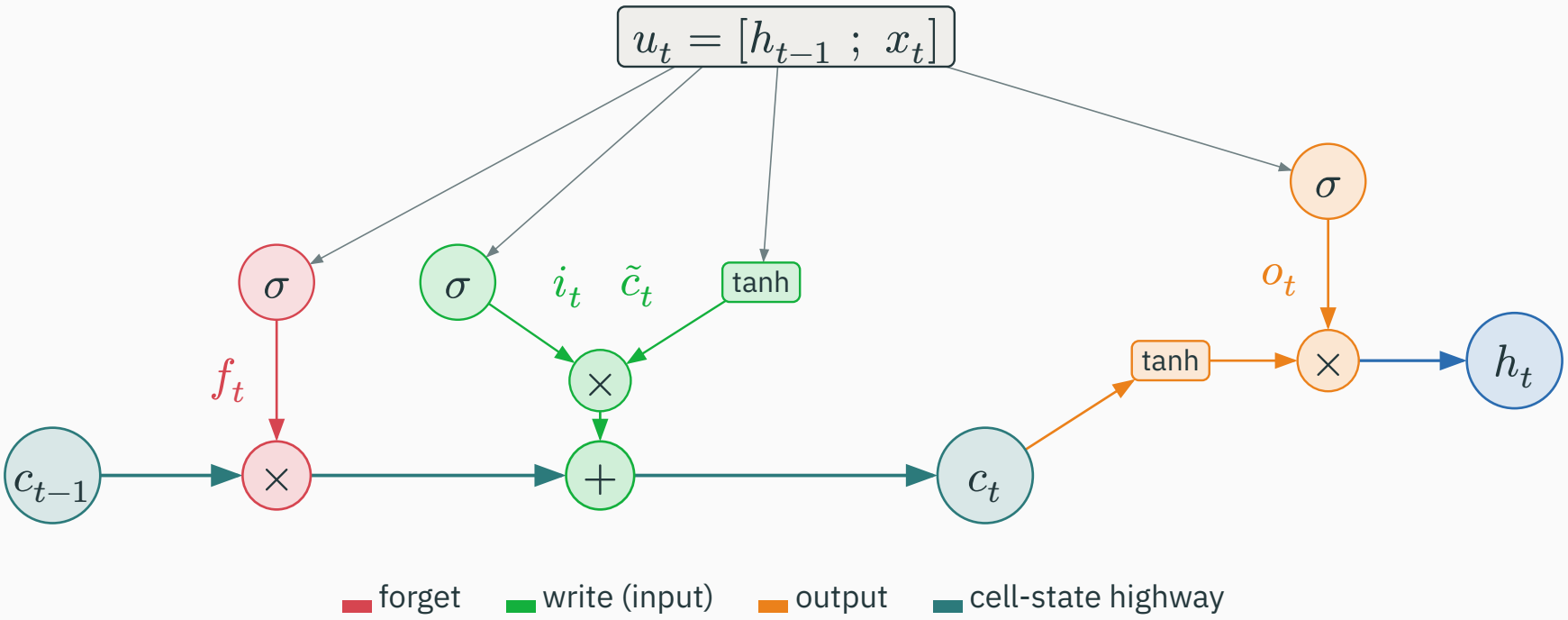
The LSTM cell

V VISUAL



The LSTM cell

V VISUAL



cell state runs straight across the top; gates tap in to forget, write, and read out

Reading the three gates

D DERIVATION

Gate	Operation	Meaning
forget f_t	$f_t \odot c_{t-1}$	1 = keep this memory, 0 = erase it
input i_t	$i_t \odot \tilde{c}_t$	how much of the new candidate to **write**
output o_t	$o_t \odot \tanh(c_t)$	how much of the memory to **expose** as h_t

Reading the three gates

D DERIVATION

Gate	Operation	Meaning
forget f_t	$f_t \odot c_{t-1}$	1 = keep this memory, 0 = erase it
input i_t	$i_t \odot \tilde{c}_t$	how much of the new candidate to **write**
output o_t	$o_t \odot \tanh(c_t)$	how much of the memory to **expose** as h_t

candidate $\tilde{c}_t = \tanh(\ldots)$ proposes **what** to write; the gates decide **how much**

Why additive memory helps

D DERIVATION

The cell state's step-to-step Jacobian **along the direct path** is just the **forget gate** — no weight matrix (the gates also depend on c_{t-1} through h_{t-1}, but this direct highway dominates):

Why additive memory helps

D DERIVATION

The cell state's step-to-step Jacobian **along the direct path** is just the **forget gate** — no weight matrix (the gates also depend on c_{t-1} through h_{t-1}, but this direct highway dominates):

$$\frac{\partial c_t}{\partial c_{t-1}} \approx \mathrm{diag}(f_t)$$

Why additive memory helps

D DERIVATION

The cell state's step-to-step Jacobian **along the direct path** is just the **forget gate** — no weight matrix (the gates also depend on c_{t-1} through h_{t-1}, but this direct highway dominates):

$$\frac{\partial c_t}{\partial c_{t-1}} \approx \operatorname{diag}(f_t)$$

So over many steps the gradient along the cell highway is a product of **gates**:

$$\frac{\partial c_T}{\partial c_k} = \prod_{j=k+1}^{T} \operatorname{diag}(f_j)$$

Why additive memory helps

D DERIVATION

The cell state's step-to-step Jacobian **along the direct path** is just the **forget gate** — no weight matrix (the gates also depend on c_{t-1} through h_{t-1}, but this direct highway dominates):

$$\frac{\partial c_t}{\partial c_{t-1}} \approx \operatorname{diag}(f_t)$$

So over many steps the gradient along the cell highway is a product of **gates**:

$$\frac{\partial c_T}{\partial c_k} = \prod_{j=k+1}^{T} \operatorname{diag}(f_j)$$

if $f_j \approx 1$, the gradient flows undamped — no repeated $\times W_{hh}$

Why additive memory helps

D DERIVATION

The cell state's step-to-step Jacobian **along the direct path** is just the **forget gate** — no weight matrix (the gates also depend on c_{t-1} through h_{t-1}, but this direct highway dominates):

$$\frac{\partial c_t}{\partial c_{t-1}} \approx \operatorname{diag}(f_t)$$

So over many steps the gradient along the cell highway is a product of **gates**:

$$\frac{\partial c_T}{\partial c_k} = \prod_{j=k+1}^{T} \operatorname{diag}(f_j)$$

if $f_j \approx 1$, the gradient flows undamped — no repeated $\times W_{hh}$

the central LSTM advantage: an **open** forget gate is a gradient highway across time

Worked scalar LSTM

D DERIVATION

One cell, old memory $c_{t-1} = 5$. **Retain** case — forget open, input nearly shut:

Worked scalar LSTM

D DERIVATION

One cell, old memory $c_{t-1} = 5$. **Retain** case — forget open, input nearly shut:

$$f_t = 0.8, \quad i_t = 0.2, \quad \tilde{c}_t = 3$$

Worked scalar LSTM

D DERIVATION

One cell, old memory $c_{t-1} = 5$. **Retain** case — forget open, input nearly shut:

$$f_t = 0.8, \quad i_t = 0.2, \quad \tilde{c}_t = 3$$

Keep a fraction of the old memory:

$$f_t \odot c_{t-1} = 0.8 \cdot 5 = 4.0$$

Worked scalar LSTM

D DERIVATION

One cell, old memory $c_{t-1} = 5$. **Retain** case — forget open, input nearly shut:

$$f_t = 0.8, \quad i_t = 0.2, \quad \tilde{c}_t = 3$$

Keep a fraction of the old memory:

$$f_t \odot c_{t-1} = 0.8 \cdot 5 = 4.0$$

Add a gated bit of the candidate:

$$i_t \odot \tilde{c}_t = 0.2 \cdot 3 = 0.6$$

Worked scalar LSTM

D DERIVATION

One cell, old memory $c_{t-1} = 5$. **Retain** case — forget open, input nearly shut:

$$f_t = 0.8, \quad i_t = 0.2, \quad \tilde{c}_t = 3$$

Keep a fraction of the old memory:

$$f_t \odot c_{t-1} = 0.8 \cdot 5 = 4.0$$

Add a gated bit of the candidate:

$$i_t \odot \tilde{c}_t = 0.2 \cdot 3 = 0.6$$

$$c_t = 4.0 + 0.6 = 4.6 \qquad \text{(memory barely moves: } 5 \to 4.6\text{)}$$

Worked scalar LSTM: replace (cont.)

D DERIVATION

Replace case — forget nearly shut, input open, candidate negative:

Worked scalar LSTM: replace (cont.)

D DERIVATION

Replace case — forget nearly shut, input open, candidate negative:

$$f_t = 0.1, \quad i_t = 0.9, \quad \tilde{c}_t = -2$$

Worked scalar LSTM: replace (cont.)

D DERIVATION

Replace case — forget nearly shut, input open, candidate negative:

$$f_t = 0.1, \quad i_t = 0.9, \quad \tilde{c}_t = -2$$

Almost none of the old memory survives:

$$f_t \odot c_{t-1} = 0.1 \cdot 5 = 0.5$$

Worked scalar LSTM: replace (cont.)

D DERIVATION

Replace case — forget nearly shut, input open, candidate negative:

$$f_t = 0.1, \quad i_t = 0.9, \quad \tilde{c}_t = -2$$

Almost none of the old memory survives:

$$f_t \odot c_{t-1} = 0.1 \cdot 5 = 0.5$$

The new candidate is written in strongly:

$$i_t \odot \tilde{c}_t = 0.9 \cdot (-2) = -1.8$$

Worked scalar LSTM: replace (cont.)

D DERIVATION

Replace case — forget nearly shut, input open, candidate negative:

$$f_t = 0.1, \quad i_t = 0.9, \quad \tilde{c}_t = -2$$

Almost none of the old memory survives:

$$f_t \odot c_{t-1} = 0.1 \cdot 5 = 0.5$$

The new candidate is written in strongly:

$$i_t \odot \tilde{c}_t = 0.9 \cdot (-2) = -1.8$$

$$c_t = 0.5 + (-1.8) = -1.3$$

Worked scalar LSTM: replace (cont.)

D DERIVATION

Replace case — forget nearly shut, input open, candidate negative:

$$f_t = 0.1, \quad i_t = 0.9, \quad \tilde{c}_t = -2$$

Almost none of the old memory survives:

$$f_t \odot c_{t-1} = 0.1 \cdot 5 = 0.5$$

The new candidate is written in strongly:

$$i_t \odot \tilde{c}_t = 0.9 \cdot (-2) = -1.8$$

$$c_t = 0.5 + (-1.8) = -1.3$$

same equation: gates near $(1, 0)$ **hold** memory ($5 \to 4.6$); near $(0, 1)$ **overwrite** it ($5 \to -1.3$)

Interactive: the LSTM cell

I INTERACTIVE

INTERACTIVE

↗ nipunbatra.github.io/interactive-articles/lstm-gates

Drag the forget, input and output gates and watch the cell state c_t retain, overwrite, or leak — and see the gradient highway open and close as f_t moves between 0 and 1.

The GRU

A simpler gated cell

The LSTM works but has **three** gates, a candidate, and **two** states. Can we do less?

A simpler gated cell

The LSTM works but has **three** gates, a candidate, and **two** states. Can we do less?

The GRU (Cho et al. 2014, in the RNN encoder–decoder for translation) keeps the gating idea but uses **one** state and **two** gates.

The GRU equations

D DERIVATION

Update gate z_t, reset gate r_t, candidate $\tilde{h}_t$, then a **convex combination**:

The GRU equations

D DERIVATION

Update gate z_t, reset gate r_t, candidate $\tilde{h}_t$, then a **convex combination**:

$$z_t = \sigma(W_z x_t + U_z h_{t-1} + b_z), \qquad r_t = \sigma(W_r x_t + U_r h_{t-1} + b_r)$$

The GRU equations

D DERIVATION

Update gate z_t, reset gate r_t, candidate $\tilde{h}_t$, then a **convex combination**:

$$z_t = \sigma(W_z x_t + U_z h_{t-1} + b_z), \qquad r_t = \sigma(W_r x_t + U_r h_{t-1} + b_r)$$

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

The GRU equations

D DERIVATION

Update gate z_t, reset gate r_t, candidate $\tilde{h}_t$, then a **convex combination**:

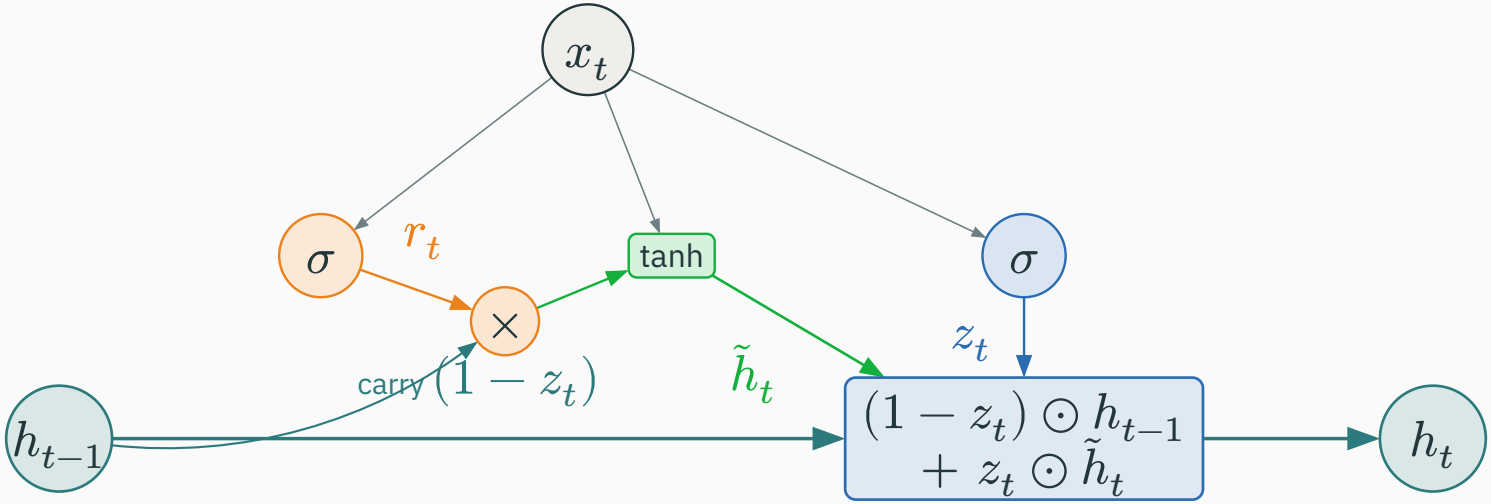
$$z_t = \sigma(W_z x_t + U_z h_{t-1} + b_z), \qquad r_t = \sigma(W_r x_t + U_r h_{t-1} + b_r)$$

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

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

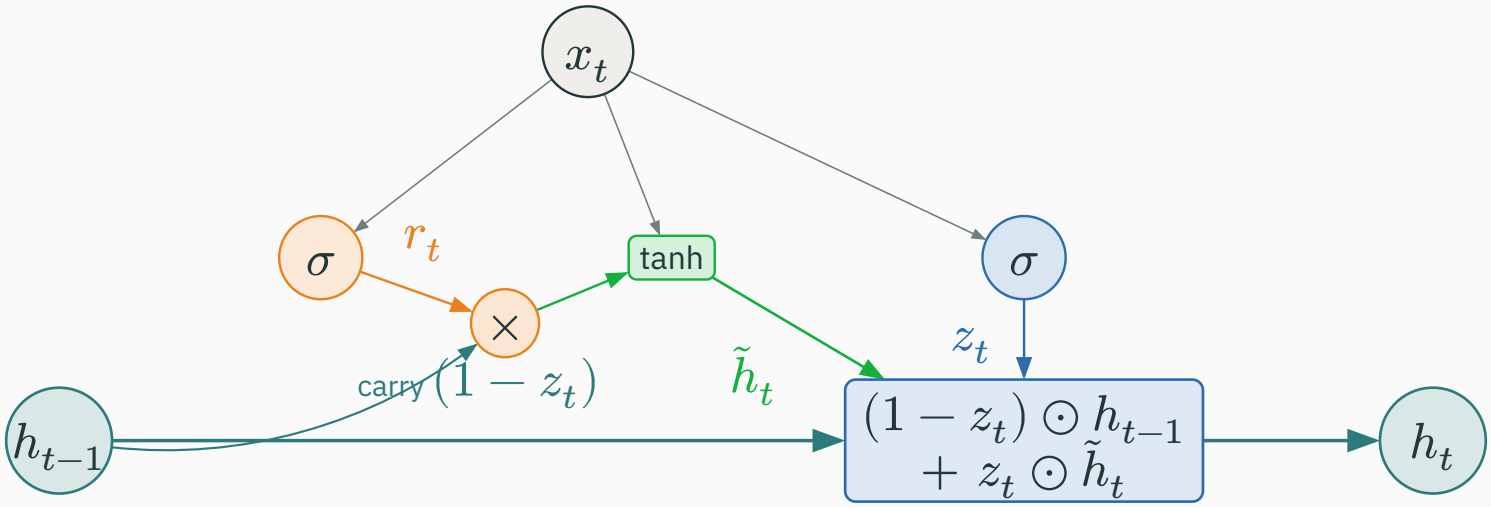
The GRU cell

V VISUAL



The GRU cell

V VISUAL



reset r_t gates the past **into** the candidate; update z_t mixes old state and candidate

Reading the two gates

D DERIVATION

Reading the two gates

D DERIVATION

update z_t: $h_t = (1 - z_t)h_{t-1} + z_t\tilde{h}_t$.
$z = 0 \to$ copy old state; $z = 1 \to$ take candidate.

reset r_t: scales h_{t-1} **inside** the candidate.
$r = 0 \to$ ignore the past, propose from x_t alone.

Reading the two gates

D DERIVATION

update z_t: $h_t = (1 - z_t)h_{t-1} + z_t\tilde{h}_t$.
$z = 0 \to$ copy old state; $z = 1 \to$ take candidate.

reset r_t: scales h_{t-1} **inside** the candidate.
$r = 0 \to$ ignore the past, propose from x_t alone.

z_t plays the LSTM's forget + input as **one** knob; $z \approx 0$ is the gradient highway

Worked scalar GRU

D DERIVATION

One unit, old state $h_{t-1} = 4$, candidate $\tilde{h}_t = 0$. **Keep** — update nearly shut:

Worked scalar GRU

D DERIVATION

One unit, old state $h_{t-1} = 4$, candidate $\tilde{h}_t = 0$. **Keep** — update nearly shut:

$$z_t = 0.25: \quad h_t = (1 - 0.25) \cdot 4 + 0.25 \cdot 0 = 0.75 \cdot 4 = 3$$

Worked scalar GRU

D DERIVATION

One unit, old state $h_{t-1} = 4$, candidate $\tilde{h}_t = 0$. **Keep** — update nearly shut:

$$z_t = 0.25: \quad h_t = (1 - 0.25) \cdot 4 + 0.25 \cdot 0 = 0.75 \cdot 4 = 3$$

Replace — update wide open:

$$z_t = 0.9: \quad h_t = (1 - 0.9) \cdot 4 + 0.9 \cdot 0 = 0.1 \cdot 4 = 0.4$$

Worked scalar GRU

D DERIVATION

One unit, old state $h_{t-1} = 4$, candidate $\tilde{h}_t = 0$. **Keep** — update nearly shut:

$$z_t = 0.25: \quad h_t = (1 - 0.25) \cdot 4 + 0.25 \cdot 0 = 0.75 \cdot 4 = 3$$

Replace — update wide open:

$$z_t = 0.9: \quad h_t = (1 - 0.9) \cdot 4 + 0.9 \cdot 0 = 0.1 \cdot 4 = 0.4$$

Check: what is h_t if $z_t = 0$ exactly?

Worked scalar GRU

D DERIVATION

One unit, old state $h_{t-1} = 4$, candidate $\tilde{h}_t = 0$. **Keep** — update nearly shut:

$$z_t = 0.25: \quad h_t = (1 - 0.25) \cdot 4 + 0.25 \cdot 0 = 0.75 \cdot 4 = 3$$

Replace — update wide open:

$$z_t = 0.9: \quad h_t = (1 - 0.9) \cdot 4 + 0.9 \cdot 0 = 0.1 \cdot 4 = 0.4$$

Check: what is h_t if $z_t = 0$ exactly?

$h_t = h_{t-1} = 4$ — the state is copied unchanged; z small holds, z large replaces ($4 \to 0.4$)

LSTM vs GRU

V VISUAL

	LSTM	GRU
states	two: h_t and c_t	one: h_t
gates	forget, input, output	update, reset
memory path	additive cell c_t, Jacobian $\mathrm{diag}(f_t)$	convex mix, carry $(1 - z_t)$
parameters	more (4 weight blocks)	fewer (3 weight blocks)

LSTM vs GRU

V VISUAL

	LSTM	**GRU**
states	two: h_t and c_t	one: h_t
gates	forget, input, output	update, reset
memory path	additive cell c_t, Jacobian $\mathrm{diag}(f_t)$	convex mix, carry $(1 - z_t)$
parameters	more (4 weight blocks)	fewer (3 weight blocks)

both fix vanishing gradients; GRU is lighter, LSTM sometimes stronger — **try both, it is task-dependent**

Interactive: the GRU cell

I INTERACTIVE

INTERACTIVE

↗ nipunbatra.github.io/interactive-articles/gru-gates

move the update and reset gates and watch the state interpolate between “keep the past” and “take the candidate”, side by side with the LSTM.

Sequence tasks with RNNs

Char-level RNN language model

D DERIVATION

Feed characters in, predict the **next** character at every step — the makemore idea, recurrent:

Char-level RNN language model

D DERIVATION

Feed characters in, predict the **next** character at every step — the makemore idea, recurrent:

t	1	2	3	4
input	a	l	i	c
target	l	i	c	e

Char-level RNN language model

D DERIVATION

Feed characters in, predict the **next** character at every step — the makemore idea, recurrent:

t	1	2	3	4
input	a	l	i	c
target	l	i	c	e

$p_t = \mathrm{softmax}(W_o h_t + b_o);$ generate by **sampling** $\hat{x}_t \sim p_t$ and feeding it back

Sequence classification

D DERIVATION

Many-to-one: read the whole sequence, predict **once**.

Sequence classification

D DERIVATION

Many-to-one: read the whole sequence, predict **once**.

last state: $\hat{y} = \mathrm{softmax}(W h_T + b)$ — the final h_T summarizes everything.

mean pool: $\hat{y} = \mathrm{softmax}\big(W\,\overline{h} + b\big)$, $\overline{h} = \frac{1}{T}\sum_t h_t$ — robust to length.

Sequence classification

D DERIVATION

Many-to-one: read the whole sequence, predict **once**.

last state: $\hat{y} = \mathrm{softmax}(W h_T + b)$ — the final h_T summarizes everything.

mean pool: $\hat{y} = \mathrm{softmax}\big(W\,\overline{h} + b\big)$, $\overline{h} = \frac{1}{T}\sum_t h_t$ — robust to length.

sentiment, topic, spam — one label from a variable-length input

Bidirectional RNNs

D DERIVATION

For **non-causal** tasks, run two RNNs — forward and backward — and concatenate:

Bidirectional RNNs

D DERIVATION

For **non-causal** tasks, run two RNNs — forward and backward — and concatenate:

$$\vec{h}_t = \mathrm{RNN}_{\rightarrow}(x_{1:t}), \qquad \overleftarrow{h}_t = \mathrm{RNN}_{\leftarrow}(x_{t:T}), \qquad h_t = \left[\vec{h}_t \; ; \; \overleftarrow{h}_t\right]$$

Bidirectional RNNs

D DERIVATION

For **non-causal** tasks, run two RNNs — forward and backward — and concatenate:

$$\vec{h}_t = \mathrm{RNN}_{\rightarrow}(x_{1:t}), \qquad \overleftarrow{h}_t = \mathrm{RNN}_{\leftarrow}(x_{t:T}), \qquad h_t = \left[\vec{h}_t \; ; \; \overleftarrow{h}_t\right]$$

Each h_t now sees the **whole** sequence — great for tagging or classification. But it peeks at the **future**, so it is **illegal for autoregressive generation**.

Padding and masking

D DERIVATION

Batches mix lengths — pad to a common T, then **mask the padding out of the loss**:

Padding and masking

D DERIVATION

Batches mix lengths — pad to a common T, then **mask the padding out of the loss**:

$$\mathcal{L} = \frac{\sum_{i,t} m_{it}\,(-\log p_{it})}{\sum_{i,t} m_{it}}, \qquad m_{it} \in \{0, 1\}$$

Padding and masking

D DERIVATION

Batches mix lengths — pad to a common T, then **mask the padding out of the loss**:

$$\mathcal{L} = \frac{\sum_{i,t} m_{it}\,(-\log p_{it})}{\sum_{i,t} m_{it}}, \qquad m_{it} \in \{0, 1\}$$

$m_{it} = 1$ for real tokens, 0 for <pad>. The mask makes padding invisible to the gradient — otherwise the model learns to predict <pad>.

Training RNNs in practice

The training loop

Detach for truncated BPTT, mask the padding, clip the gradient:

The training loop

Detach for truncated BPTT, mask the padding, clip the gradient:

```
h = model.init_hidden(batch_size)
for x, y, mask in loader:
    h = h.detach()                      # truncated BPTT: cut the graph
    logits, h = model(x, h)             # forward one chunk
    loss = masked_cross_entropy(logits, y, mask)   # ignore <pad>
    optimizer.zero_grad()
    loss.backward()                     # BPTT within the chunk
    nn.utils.clip_grad_norm_(model.parameters(), 5.0)  # tame explosions
    optimizer.step()
```

The training loop

Detach for truncated BPTT, mask the padding, clip the gradient:

```
h = model.init_hidden(batch_size)
for x, y, mask in loader:
    h = h.detach()                      # truncated BPTT: cut the graph
    logits, h = model(x, h)             # forward one chunk
    loss = masked_cross_entropy(logits, y, mask)   # ignore <pad>
    optimizer.zero_grad()
    loss.backward()                     # BPTT within the chunk
    nn.utils.clip_grad_norm_(model.parameters(), 5.0)  # tame explosions
    optimizer.step()
```

the three RNN-specific lines: `detach`, `masked` loss, `clip_grad_norm_`

Common bugs

V VISUAL

Symptom	Likely cause
loss goes `NaN`	exploding gradients — clip; lower the learning rate
predicts `<pad>` a lot	no loss mask on padded positions
memory grows each step	state not detached — full graph kept forever
no long-term dependence	vanilla RNN vanishing — use LSTM / GRU
generation repeats itself	greedy decoding — sample / temperature / top-k
training is slow	not packing / batching variable lengths

RNN vs TCN — a preview

The RNN is inherently **sequential**: h_t needs h_{t-1}, so time cannot be parallelized.

RNN vs TCN — a preview

The RNN is inherently **sequential**: h_t needs h_{t-1}, so time cannot be parallelized.

RNN: $h_t = f(h_{t-1}, x_t)$ — a state carried step by step; $O(T)$ sequential.

TCN: stacked **dilated causal convolutions** — a fixed receptive field, fully **parallel** over time.

RNN vs TCN — a preview

The RNN is inherently **sequential**: h_t needs h_{t-1}, so time cannot be parallelized.

RNN: $h_t = f(h_{t-1}, x_t)$ — a state carried step by step; $O(T)$ sequential.

TCN: stacked **dilated causal convolutions** — a fixed receptive field, fully **parallel** over time.

next lecture: convolutions over time, and then **attention** removes the recurrence entirely

Summary

One-slide summary

V VISUAL

Idea	**Takeaway**
recurrent state	$h_t = f_\theta(h_{t-1}, x_t)$ — shared f_θ, any length, memory in h_t
BPTT	unroll in time; shared-weight gradients **accumulate** over steps
vanilla RNN	$\partial h_T / \partial h_k \approx w^T$ — vanishing / exploding gradients
clipping	caps exploding norm; does **nothing** for vanishing
LSTM	cell $c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t$; forget / input / output gates
GRU	one state, update + reset gates; simpler, often as good

Final mental model — one idea

RNNs **remember** by carrying a hidden state.

LSTMs and GRUs **learn when** to remember, forget, and update.

a gate is a controllable path — for information **and** for gradients

We gave the model a **memory** and gates to control it.

Next: **attention and seq2seq** — let every step look **directly** at every other, no recurrence required.