Sequence Models as Dynamical Systems

Mathematics of Deep Learning — Chapter 9

Yin Yang

Foundation Books Project

Roadmap

Sequence Data and Order

Embeddings to Hidden States

The Vanilla RNN

Discrete-Time Dynamical Systems

Outputs, Losses, Padding, Masks

Unrolling the Computation Graph

Backpropagation Through Time

Vanishing and Exploding Gradients

Gated Recurrences

Practical Variants

Limits and Attention Bridge

Summary

Sequence Data and Order

Why Sequences Are Different

Earlier chapters treated data as single objects: an image, a vector, a feature row. Sequences add one ingredient: *order*.

- Running example: the phrase `not very good`.
- Same three words as `good very not`, but different meaning.
- Order is part of the input, not an afterthought.

After tokenization and embedding lookup:

$$x_1,\ x_2,\ x_3 \in \mathbb{R}^{d_x}, \qquad x_1 = \texttt{not},\ x_2 = \texttt{very},\ x_3 = \texttt{good}.$$

The subscript t is a *time* index, not an exponent.

A sequence is an ordered list; bag-of-words models throw the order away.

Finite Sequences, Prefixes, Map Types

Finite sequence of length T
Ordered list $x_{1:T} = (x_1, \ldots, x_T)$ with each $x_t \in \mathbb{R}^{d_x}$.

Prefix through time t
$x_{1:t} = (x_1, \ldots, x_t)$. A *causal* model at time t may use $x_{1:t}$, never $x_{t+1:T}$.

Three common map types:

$$f : \mathbb{R}^{T \times d_x} \to \mathcal{Y}, \quad f : \mathbb{R}^{T \times d_x} \to \mathcal{Y}^T, \quad f : \mathbb{R}^{T \times d_x} \to \mathbb{R}^{T \times d_y}.$$

Sequence-to-one (sentiment), sequence-to-sequence (tagging, next-token).

First decide: causal or offline? That decision constrains architecture, loss, and use.

Embeddings to Hidden States

From Embeddings to a Hidden State

The embedding x_t represents *one item*. It does not remember the prefix.

Hidden state
For sequence $x_{1:T}$, a hidden state is

$$h_t \in \mathbb{R}^{d_h}$$

computed after the model has processed $x_{1:t}$. Initial state h_0 is fixed (often 0) or learned.

General recurrent update, parameters θ shared across time:

$$h_t = F_\theta(h_{t-1}, x_t).$$

Same θ at $t = 1, 2, \ldots, T$ — the time analogue of convolutional weight sharing.

The hidden state is *learned* task-specific memory — never supplied as a label.

A Tiny Scalar Memory Rule

Suppose a toy model uses scalar inputs and scalar states:

$$h_t = 0.5\, h_{t-1} + x_t, \qquad h_0 = 0.$$

For $x_1 = 2,\ x_2 = -1,\ x_3 = 3$:

$$h_1 = 2, \qquad h_2 = 0, \qquad h_3 = 3.$$

- The state is *not* just the latest input.
- It is the result of repeatedly applying the same rule.
- A fixed-size h_t must compress a possibly long prefix into d_h numbers.

Compression into a fixed-size state is useful — and is the root of long-range dependency problems later.

The Vanilla RNN

Vanilla RNN Update

Definition
For $x_t \in \mathbb{R}^{d_x}$, $h_{t-1} \in \mathbb{R}^{d_h}$,

$$a_t = W_x x_t + W_h h_{t-1} + b, \qquad h_t = \phi(a_t),$$

with $W_x \in \mathbb{R}^{d_h \times d_x}$, $W_h \in \mathbb{R}^{d_h \times d_h}$, $b \in \mathbb{R}^{d_h}$. Activation ϕ is elementwise (tanh or ReLU).

Shape check: both $W_x x_t$ and $W_h h_{t-1}$ lie in $\mathbb{R}^{d_h}$, so their sum is a valid preactivation. h_t has the same dimension as h_{t-1} — ready for the next step.

Recurrent parameter count:

$$d_h d_x + d_h^2 + d_h \quad \text{(independent of } T\text{)}.$$

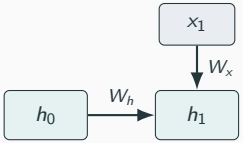
The state dimension must stay fixed so the recurrence can repeat.

Unrolling the RNN, Step by Step



(1) Start at the initial state h_0.

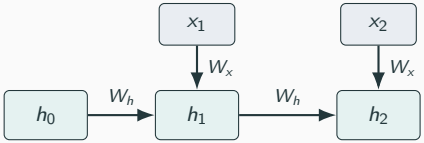
Unrolling the RNN, Step by Step



(1) Start at the initial state h_0.

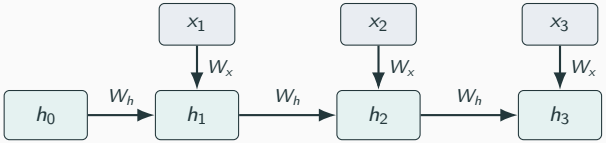
(2) Read x_1 and update to h_1.

Unrolling the RNN, Step by Step



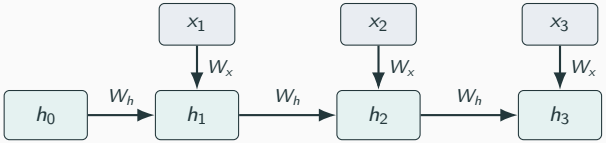
(1) Start at the initial state h_0.
(2) Read x_1 and update to h_1.
(3) Read x_2 and update to h_2.

Unrolling the RNN, Step by Step



(1) Start at the initial state h_0.
(2) Read x_1 and update to h_1.
(3) Read x_2 and update to h_2.
(4) Read x_3 and update to h_3.

Unrolling the RNN, Step by Step



Same W_x, W_h, b are reused at every time step. Parameter count is independent of sequence length.

(1) Start at the initial state h_0.
(2) Read x_1 and update to h_1.
(3) Read x_2 and update to h_2.
(4) Read x_3 and update to h_3.
(5) The same parameters appear at every transition.

Unrolling makes a single recurrence look like a feedforward graph with tied weights.

One Scalar RNN Step

Take $d_x = d_h = 1$, $\phi = \tanh$, $W_x = 2$, $W_h = 0.5$, $b = -1$, $h_0 = 0$, $x_1 = 1$:

$$a_1 = 2(1) + 0.5(0) - 1 = 1, \qquad h_1 = \tanh(1) \approx 0.762.$$

Parameter count example: $d_x = 12$, $d_h = 8$,

$$8 \cdot 12 + 8^2 + 8 = 96 + 64 + 8 = 168 \text{ scalars.}$$

A length-50 sequence applies the update 50 times but uses the same 168 scalars.

Length affects how often the rule fires — not how many parameters the rule has.

Discrete-Time Dynamical Systems

RNNs as Discrete-Time Dynamical Systems

The recurrence

$$h_t = F_\theta(h_{t-1}, x_t)$$

is a discrete-time dynamical system driven by inputs.

Trajectory
Given h_0 and $x_{1:T}$, the trajectory is the ordered list $h_0, h_1, \ldots, h_T$.

Remove the input and the nonlinearity. Scalar linear case:

$$h_t = u\, h_{t-1} \implies h_t = u^t h_0.$$

Memory, decay, oscillation, and instability all originate from repeating one update.

Four Scalar Regimes, Built Up

$|u| < 1$
decay
$h_t \to 0$

(1) Below unit magnitude: the state forgets exponentially.

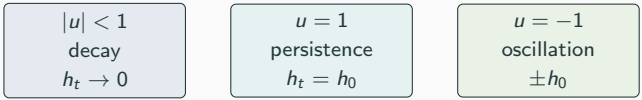
Four Scalar Regimes, Built Up

$|u| < 1$
decay
$h_t \to 0$

$u = 1$
persistence
$h_t = h_0$

(1) Below unit magnitude: the state forgets exponentially.
(2) Unit: perfect memory of the initial state.

Four Scalar Regimes, Built Up



(1) Below unit magnitude: the state forgets exponentially.
(2) Unit: perfect memory of the initial state.
(3) Negative unit: alternating sign without magnitude growth.

Four Scalar Regimes, Built Up

$\lvert u\rvert < 1$	$u = 1$	$u = -1$	$\lvert u\rvert > 1$
decay	persistence	oscillation	explosion
$h_t \to 0$	$h_t = h_0$	$\pm h_0$	$\lvert h_t\rvert \to \infty$

(1) Below unit magnitude: the state forgets exponentially.
(2) Unit: perfect memory of the initial state.
(3) Negative unit: alternating sign without magnitude growth.
(4) Above unit magnitude: any nonzero state blows up.

These four regimes are the scalar shadows of vanishing, memory, oscillation, and exploding signals.

Spectral Tools and Fixed Points

Vector case: if $Wv = \lambda v$, then $W^t v = \lambda^t v$. Different eigen-directions can scale by different amounts.

- Spectral radius (largest $|\lambda|$): warning light for growth or decay along special directions.
- Spectral norm $\|W\|_2$ (largest singular value): safe upper bound on stretching, satisfies $\|AB\|_2 \leq \|A\|_2 \|B\|_2$.

Fixed point
For input-free $h_t = F_\theta(h_{t-1})$, a state $h^\star$ with $F_\theta(h^\star) = h^\star$. Locally stable if nearby trajectories return to it.

Near $h^\star$, the Jacobian $J = \partial F_\theta / \partial h$ plays the role of the linear update.

Spectral quantities summarize how repeated multiplication shrinks, preserves, or amplifies different directions.

Outputs, Losses, Padding, Masks

Where Outputs Attach

Many-to-one (e.g. sentiment): predict from final state

$$\hat{y} = g_\psi(h_T), \qquad p = \mathrm{softmax}(W_y h_T + c).$$

Many-to-many (e.g. tagging, next-token):

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

Same hidden states, different read-out points.

The task decides which hidden states carry losses; the recurrence stays the same.

Two-Step Cross-Entropy Example

Three-class tagger logits:

$$s_1 = (2, 1, 0), \qquad s_2 = (0, 1, 2).$$

Softmax probabilities:

$$\text{softmax}(s_1) \approx (0.665, 0.245, 0.090),$$

$$\text{softmax}(s_2) \approx (0.090, 0.245, 0.665).$$

Correct classes 1 and 3, average cross-entropy:

$$\frac{-\log(0.665) - \log(0.665)}{2} \approx 0.408.$$

Cross-entropy is averaged over the *real* output positions — not array slots.

What counts as a "position" is a modelling choice, not an array shape.

Padding and Masks

Padding mask
For padded length $T_{\max}$, a vector $m_1, \ldots, m_{T_{\max}} \in \{0, 1\}$ with $m_t = 1$ for real tokens and $m_t = 0$ for padding.

Masked sequence loss

$$\mathcal{L}_{\text{masked}} = \frac{\sum_{t=1}^{T_{\max}} m_t \, \ell(\hat{y}_t, y_t)}{\sum_{t=1}^{T_{\max}} m_t}.$$

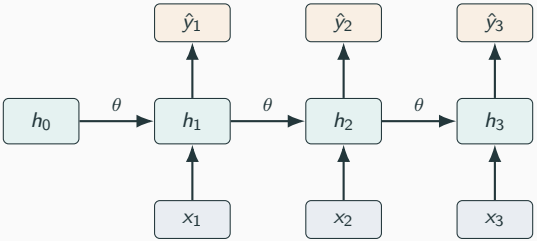
Concrete: per-step losses $0.2, 0.4, 1.0, 9.0$ with mask $(1, 1, 1, 0)$:

$$\text{unmasked} = \tfrac{10.6}{4} = 2.65, \qquad \text{masked} = \tfrac{1.6}{3} \approx 0.533.$$

The mask is not an implementation detail — it defines which terms belong in the empirical risk.

Unrolling the Computation Graph

The Unrolled Graph and Shared Parameters



If parameters were untied as $\theta_1, \ldots, \theta_T$, tying imposes $\theta_1 = \cdots = \theta_T = \theta$, so

$$\nabla_\theta L = \sum_{t=1}^{T} \nabla_{\theta_t} L \Big|_{\theta_1 = \cdots = \theta_T = \theta}.$$

Sharing means one perturbation hits every time step — gradients add.

Backpropagation Through Time

BPTT: The Recursion

Many-to-many loss $L = \sum_t \ell_t$, vanilla RNN $a_t = W_x x_t + W_h h_{t-1} + b$, $h_t = \phi(a_t)$.

Define

$$q_t = \frac{\partial L}{\partial h_t}, \quad \delta_t = \frac{\partial L}{\partial a_t}, \quad r_t = \frac{\partial \ell_t}{\partial h_t}.$$

Vanilla RNN BPTT
With $\delta_{T+1} = 0$, for $t = T, T-1, \ldots, 1$:

$$q_t = r_t + W_h^\top \delta_{t+1}, \qquad \delta_t = q_t \odot \phi'(a_t).$$

Two sources at every h_t: local output loss r_t and future signal $W_h^\top \delta_{t+1}$.

BPTT is just the chain rule, walked backward through the unrolled graph.

Shared-Parameter Gradients

Once δ_t are known, parameters sum their contributions:

$$\frac{\partial L}{\partial W_x} = \sum_{t=1}^{T} \delta_t x_t^\top, \quad \frac{\partial L}{\partial W_h} = \sum_{t=1}^{T} \delta_t h_{t-1}^\top, \quad \frac{\partial L}{\partial b} = \sum_{t=1}^{T} \delta_t.$$

Three-step scalar example: $h_t = \tanh(wx_t + uh_{t-1} + b)$, $h_0 = 0$, final-state loss only, $e = \hat{y} - y$. Backward:

$$q_3 = ve, \quad \delta_3 = q_3(1 - h_3^2), \quad q_2 = u\delta_3, \quad \delta_2 = q_2(1 - h_2^2), \quad \ldots$$

$$\frac{\partial L}{\partial w} = \delta_1 x_1 + \delta_2 x_2 + \delta_3 x_3, \quad \frac{\partial L}{\partial u} = \delta_2 h_1 + \delta_3 h_2.$$

Shared weights collect one contribution per use — the time-axis analogue of convolutional weight sharing.

Truncated BPTT

Full BPTT is expensive: long graph, must store forward states.

Truncated BPTT (window K)
Backpropagate through at most K recurrent transitions; carry forward state past the boundary but treat the backward graph as constant there.

- Exact gradient for the truncated graph.
- Biased relative to the full BPTT gradient: long-range terms are dropped.
- Works best when the task mostly needs recent context.
- Padded positions must not create artificial signals.

Truncation is a memory/compute tradeoff that quietly shortens what gradients can see.

Vanishing and Exploding Gradients

The Temporal Gradient Product

Chain rule across time:

$$\frac{\partial h_T}{\partial h_t} = \frac{\partial h_T}{\partial h_{T-1}} \frac{\partial h_{T-1}}{\partial h_{T-2}} \cdots \frac{\partial h_{t+1}}{\partial h_t}.$$

One Jacobian factor per time step between t and T.

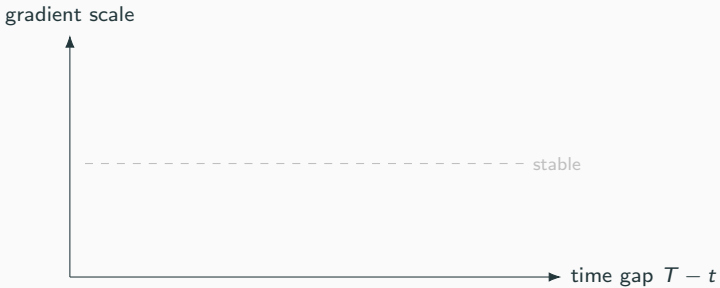
Norm bound
If $\|J_s\|_2 \leq \rho$ for $s = t+1, \ldots, T$, then

$$\left\| \frac{\partial h_T}{\partial h_t} \right\|_2 \leq \rho^{T-t}.$$

Scalar version: $\partial h_T / \partial h_t = u^{T-t}$. $u = 0.5$, $T - t = 10$: derivative ≈ 0.001. $u = 1.5$: derivative ≈ 57.7.

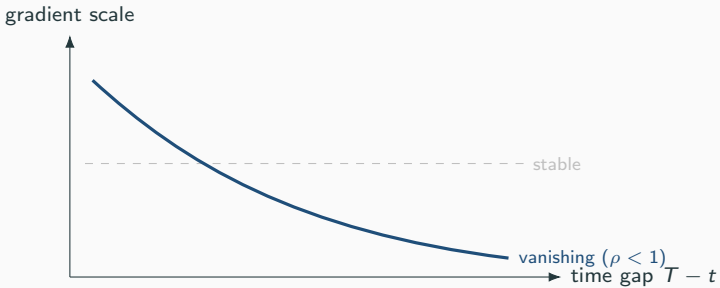
Long-range learning is a battle with a long product of Jacobian factors.

Vanishing vs. Exploding, Animated



(1) Baseline: gradient magnitude on the vertical axis vs. time gap on the horizontal axis.

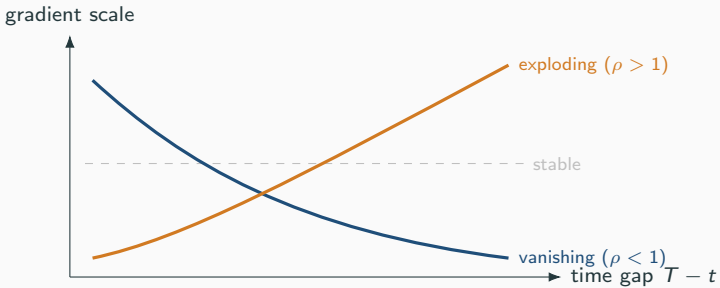
Vanishing vs. Exploding, Animated



(1) Baseline: gradient magnitude on the vertical axis vs. time gap on the horizontal axis.

(2) Repeated factors of norm < 1 erase distant credit assignment.

Vanishing vs. Exploding, Animated



(1) Baseline: gradient magnitude on the vertical axis vs. time gap on the horizontal axis.
(2) Repeated factors of norm < 1 erase distant credit assignment.
(3) Repeated factors of norm > 1 produce destabilising updates.

Clipping caps the magnitude; it does not heal vanishing gradients.

Gradient Clipping

Norm clipping (threshold $c > 0$)

$$g_{\text{clip}} = \begin{cases} g, & \|g\| \le c, \\ c\, g / \|g\|, & \|g\| > c. \end{cases}$$

- Prevents single very large updates from destabilising training.
- Does *not* fix vanishing gradients — those require an architectural change.
- Cheap, universal, almost always applied alongside RNNs.

Clipping treats the symptom of explosion — gates treat the cause of vanishing.

Gated Recurrences

Gates as Learned Interpolation

A *gate* is a vector $z \in (0,1)^{d_h}$, usually produced by a sigmoid layer.

Elementwise interpolation:

$$z \odot a + (1 - z) \odot b.$$

GRU-style update:

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

Coordinate j: if $(z_t)_j \approx 1$ keep the old value; if $(z_t)_j \approx 0$ overwrite with the candidate.

Worked numbers: $h_{t-1} = 0.8$, $\widetilde{h}_t = -0.2$:

$$z = 0.9 \Rightarrow h_t = 0.70, \qquad z = 0.1 \Rightarrow h_t = -0.10.$$

Gates let the network *learn* when to remember and when to overwrite.

Gated Update, Built Up

old state
h_{t-1}

(1) Carry the old state in as one input.

Gated Update, Built Up

old state
h_{t-1}

candidate
$\widetilde{h}_t$

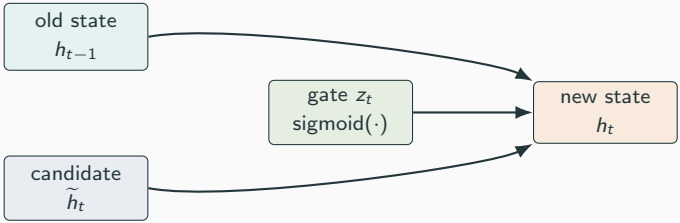
(1) Carry the old state in as one input.
(2) Compute a candidate from x_t and h_{t-1}.

Gated Update, Built Up



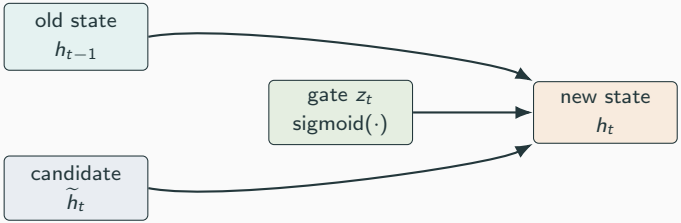
(1) Carry the old state in as one input.
(2) Compute a candidate from x_t and h_{t-1}.
(3) A sigmoid layer outputs the gate $z_t \in (0,1)^{d_h}$.

Gated Update, Built Up



(1) Carry the old state in as one input.
(2) Compute a candidate from x_t and h_{t-1}.
(3) A sigmoid layer outputs the gate $z_t \in (0,1)^{d_h}$.
(4) The new state interpolates old and candidate.

Gated Update, Built Up



Direct path:
$\partial h_t / \partial h_{t-1} = \mathrm{diag}(z_t)$
when $z_t, \widetilde{h}_t$ are fixed.

(1) Carry the old state in as one input.
(2) Compute a candidate from x_t and h_{t-1}.
(3) A sigmoid layer outputs the gate $z_t \in (0,1)^{d_h}$.
(4) The new state interpolates old and candidate.
(5) Direct memory path has a diagonal Jacobian, not a full W_h.

A near-copy path lets gradients survive across many steps.

LSTM Cell: An Additive Memory Path

An LSTM separates hidden state h_t from cell state c_t:

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

Gates: forget f_t, input i_t, output o_t, all sigmoid; candidate $\widetilde{c}_t = \tanh(\cdot)$.

Direct cell-state path (gates fixed):

$$\frac{\partial c_T}{\partial c_t} = \operatorname{diag}(f_T)\,\operatorname{diag}(f_{T-1}) \cdots \operatorname{diag}(f_{t+1}).$$

Coordinate j: $\prod_{s=t+1}^{T} (f_s)_j$. If forget gates stay near 1, gradient *copies* across time.

The additive cell-state path is the key change: a learned near-identity instead of a repeated W_h.

One Scalar LSTM Backward Step

Given $\bar{h}_t = 2$, $o_t = 0.5$, $c_t = 0$, $\bar{c}_{t+1} = 3$, $f_{t+1} = 0.8$:

$$\bar{c}_t = \bar{h}_t\, o_t(1 - \tanh^2(c_t)) + \bar{c}_{t+1}\, f_{t+1} = 2(0.5)(1) + 3(0.8) = 3.4.$$

Continue with $c_{t-1} = 1.5$, $i_t = 0.25$, $\tilde{c}_t = -0.4$:

$$\bar{f}_t = 3.4(1.5) = 5.1, \quad \bar{i}_t = 3.4(-0.4) = -1.36, \quad \bar{\tilde{c}}_t = 3.4(0.25) = 0.85.$$

Output-gate gradient $\bar{o}_t = \bar{h}_t \tanh(c_t) = 0$ since $c_t = 0$.

Two contributions to $\bar{c}_t$: local through tanh and direct copy from $\bar{c}_{t+1}$.

The forget gate scales the future signal — one multiplication per step, no recurrent matrix.

Practical Variants

Bidirectional and Stacked Recurrences

Bidirectional

Forward state $\overrightarrow{h}_t$ from $x_{1:t}$, backward state $\overleftarrow{h}_t$ from $x_{T:t}$, concatenated:

$$h_t = \begin{bmatrix} \overrightarrow{h}_t \\ \overleftarrow{h}_t \end{bmatrix}.$$

Not causal for $t < T$: depends on future inputs.

Stacked

$$h_t^{(1)} = F_{\theta_1}(h_{t-1}^{(1)}, x_t), \qquad h_t^{(2)} = F_{\theta_2}(h_{t-1}^{(2)}, h_t^{(1)}).$$

Layer 1 processes inputs; layer 2 processes layer-1 representations.

Bidirectional is a graph choice; stacking is depth in the state-update composition.

Limits and Attention Bridge

Limits of Recurrence

- **Bottleneck:** a many-to-one RNN packs $x_{1:T}$ into a fixed $h_T \in \mathbb{R}^{d_h}$.
- **Path length:** information from x_1 travels $O(T)$ state transitions to reach h_T; gradients travel back the same way.
- **Sequential compute:** h_t depends on h_{t-1} — no parallelism along time.

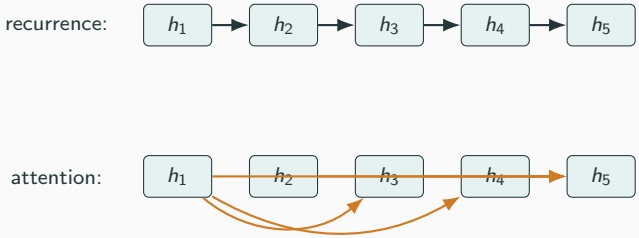
In graph terms:

$$x_1 \to h_1 \to h_2 \to \cdots \to h_T,$$

length $\propto T$, so $\partial h_T / \partial h_1$ is a long Jacobian product.

Long chains create long gradient products, which is exactly where vanishing and explosion live.

Bridge to Attention



- Recurrence: chain of state transitions, length $O(T)$.
- Attention: direct comparisons between positions, path length $O(1)$ per layer.
- Algebra of queries, keys, and values comes next chapter.

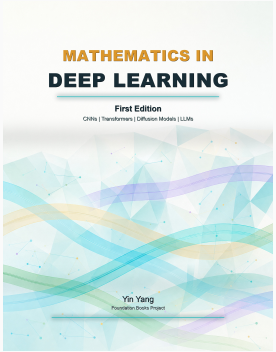
Attention does not erase the ideas of state and sharing — it changes the graph along which they travel.

Summary

Chapter in One Slide

- **Sequences** are ordered; causality decides whether future tokens are allowed.
- **Hidden states** are learned, fixed-size, task-specific memory; updated by a shared $F_\theta(h_{t-1}, x_t)$.
- **Vanilla RNN:** affine + nonlinearity; parameter count $d_h d_x + d_h^2 + d_h$, independent of T.
- **Dynamical systems view:** scalar u^t gives memory, decay, oscillation, or explosion; spectra extend this to matrices.
- **Losses, padding, masks:** the mask defines which terms belong in $\widehat{R}$.
- **BPTT:** chain rule on the unrolled graph; shared parameters sum contributions across time.
- **Vanishing / exploding** comes from products of Jacobians; clipping caps explosion, gates ease vanishing.
- **Gates / LSTMs** give a learned near-copy memory path. **Attention** shortens the path further.

Next: attention as algebra over queries, keys, and values — and the architectures that replaced recurrent backbones.



Mathematics in Deep Learning

Chapter overview slides for the book by Yin Yang.

Book page	`foundation-books.github.io/mathematics-in-deep-learning`
Code	`github.com/foundation-books/mathematics-in-deep-learning-code`
Paperback	`amazon.com/dp/B0H1ZZHDT2`
Hardcover	`amazon.com/dp/B0H1ZL11MJ`
Kindle	`amazon.com/dp/B0GX32M8SP`
License	CC BY-NC-ND 4.0

These free overview slides may be shared non-commercially with attribution and without modifications. Full explanations, exercises, and companion-code context are in the book and linked repository.