Lecture 20: Recurrent Neural Networks

Tao Huang

John Hopcroft Center, School of Computer Science, Shanghai Jiao Tong University

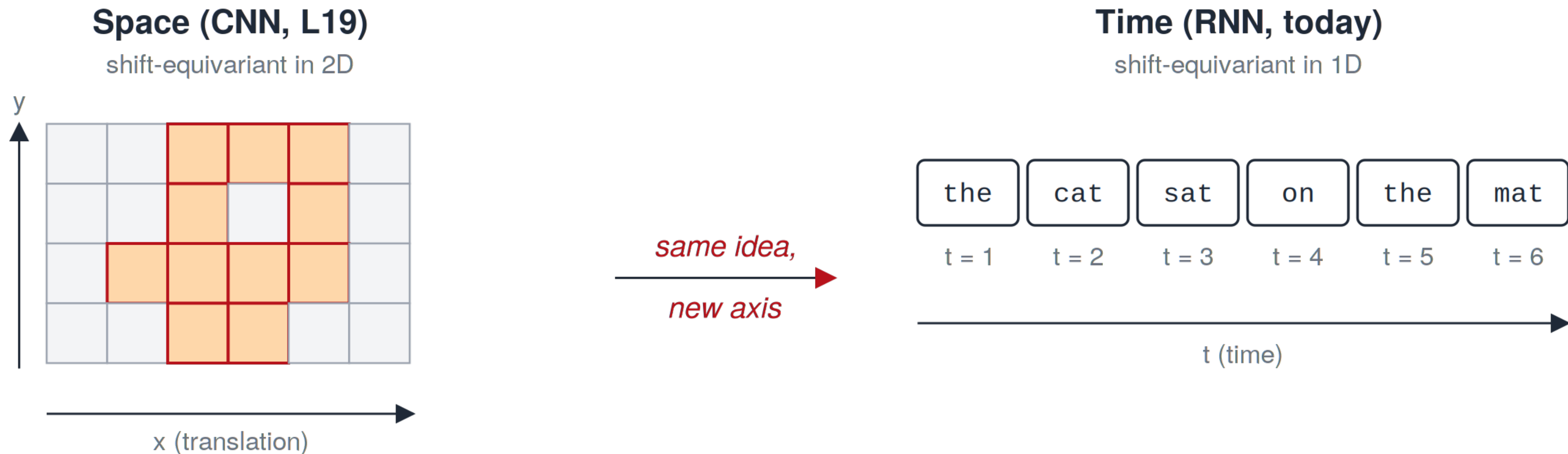
https://taohuang.info/cs3317

https://oc.sjtu.edu.cn/courses/89538

AI tools assisted in generating some figures in these slides. All such content has been reviewed, and the instructor is responsible for its accuracy.

From Space to Time

- Last week, CNNs (L17–19) gave us a strong inductive bias for space: shift-equivariance over a 2D grid.
  But text, audio, and video unfold over time — a different axis with different rules.
- Today we ask: **what is the protocon for sequences?**

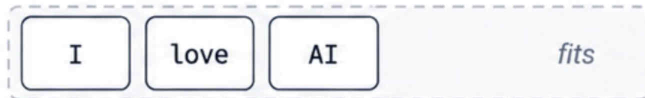


Why an MLP (or even a CNN) fails on sequences

- A vanilla MLP forces a fixed input length. A CNN handles variable length but not arbitrary range. Three concrete failures motivate a new primitive:

1. Variable length

"I love AI" (3 tokens)

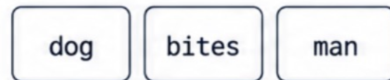


"I really love studying AI at SJTU" (8)

overflow

2. Order matters

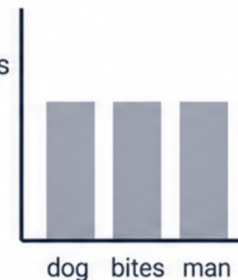
"dog bites man"



"man bites dog"

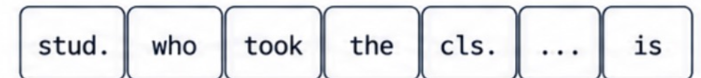
bag-of-words →

identical histogram



3. Long-range dependency

"The student who took the class ... is graduating."



subject-verb agreement

CNN receptive field would need ~8 layers to span this

Objectives

After this lecture, you should be able to:

- Derive the vanilla RNN cell and write its forward pass on a whiteboard.
- Diagnose vanishing/exploding gradients from the spectral radius of the recurrent Jacobian.
- Explain why the LSTM cell-state highway preserves gradient flow — and why it is the same trick as a ResNet skip connection.
- Choose between LSTM, GRU, and (a peek ahead) Transformer / SSM for a given task.
- Identify the structural walls — sequential compute, bottleneck, finite memory — that motivate everything in L21–L22.

1. From CNN to RNN

Building the cell from three requirements

Suppose we want a network that reads tokens $x_1, x_2, \dots, x_T$ one at a time and produces y_t at each step. What must hold?

- **Requirement 1 —** handle any *T*: parameters cannot depend on length.
- **Requirement 2 —** respect order: y_t must depend on the past, not the future.
- **Requirement 3 —** share statistical strength: the rule learned at *t = 7* should help at *t = 700*.

Consequence: introduce a fixed-size **hidden state** h_t that summarises everything seen so far, and a **single function** *f* — with shared parameters — that updates it. This forces:

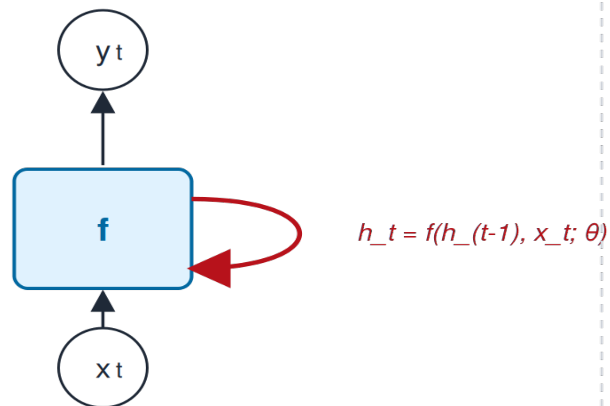
$$h_t = f(h_{t-1}, x_t; \theta), \quad y_t = g(h_t; \theta_y)$$ with the same θ at every step.

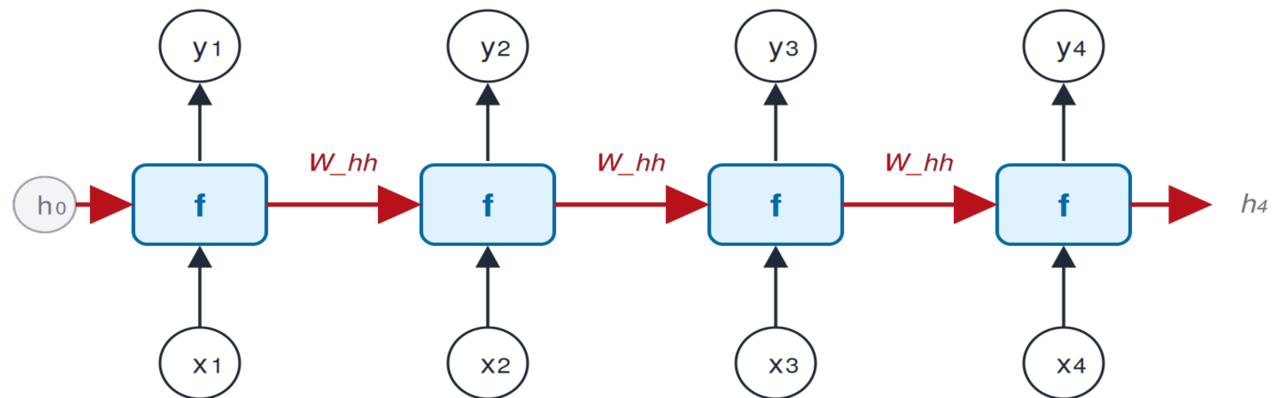
The vanilla RNN: rolled and unrolled views

Concrete instantiation (Elman, 1990):

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t + b_h)\,, \quad y_t = W_{hy}h_t + b_y$$

- The **rolled** diagram shows the self-loop — the definition. The unrolled diagram is the same network drawn at each time step, sharing parameters. This is the picture you backprop through.





Quick check — count the parameters

Question: for hidden size $d = 256$, input size $n = 100$, and a sequence of length $T = 50$, how many learnable parameters does the vanilla RNN have? How does that change with T?

Pause. Try it before reading on.

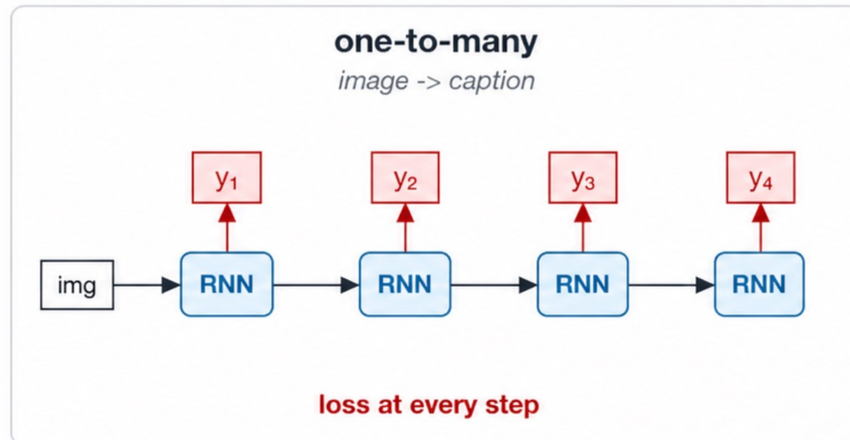
Answer: W_{hh} is $d \times d$ = 65,536; W_{xh} is $d \times n$ = 25,600; W_{hy} and biases negligible by comparison. Total ≈ **90 K parameters** — independent of T.

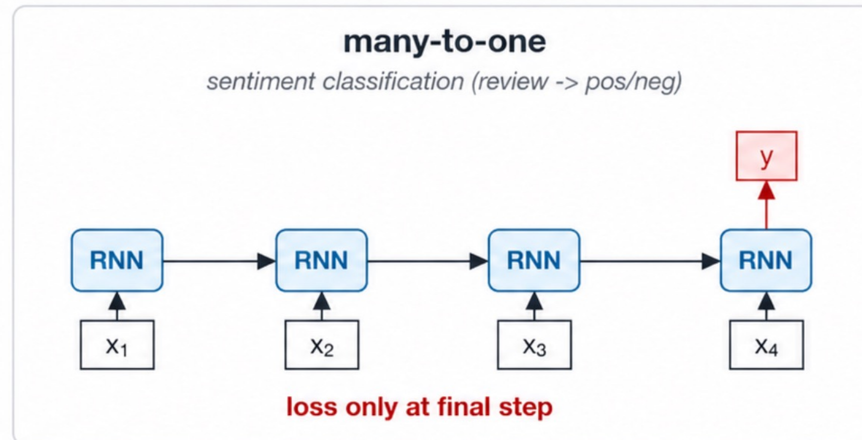
Key insight: parameter count is **constant in sequence length**. This is the payoff of weight sharing — and the reason RNNs train at all.

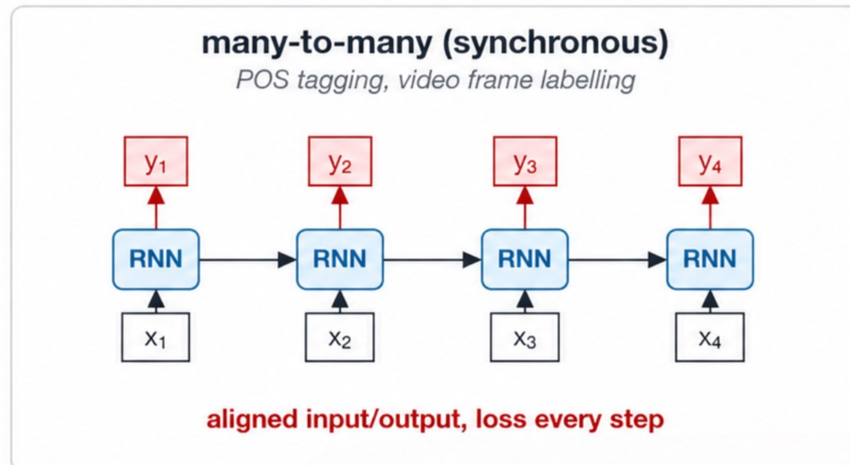
Follow-up: compute, on the other hand, scales as $O(T \cdot d^2)$. Memory for BPTT scales as $O(T \cdot d)$. Remember these — they motivate truncated BPTT (slide 11) and the sequential-compute wall (slide 25).

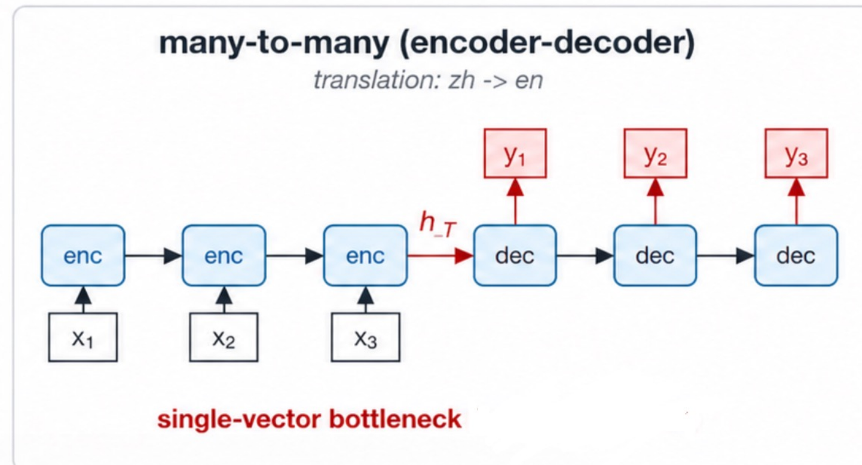
Four sequence patterns the same cell can express

- Same RNN cell, different input/output wiring — that is the source of RNNs' versatility.









2. Training: BPTT & vanishing gradients

The vanishing/exploding gradient — concretely

Computing $\partial L_T / \partial h_1$ means multiplying T Jacobians of the recurrent map. If their typical norm is ρ:

- if $\rho < 1$ → product → 0 → nothing learned across long ranges
- if $\rho > 1$ → product → ∞ → NaN training

This is not a tuning bug. It is a structural feature of multiplicative chains.

Vanishing - every Jacobian factor < 1

0.6 x 0.4 x 0.7 x 0.3 x 0.5 x ... x 0.4 --(~= 0.5^100)--> ~= 7.9 x 10^(-31)

100 factors, each ||dh_t/dh_(t-1)|| ~= 0.5

gradient at h_1 from L_(100): vanished

Exploding - every Jacobian factor > 1

1.4 x 1.6 x 1.3 x 1.5 x 1.4 x ... x 1.5 --(~= 1.45^100)--> ~= 1.7 x 10^16

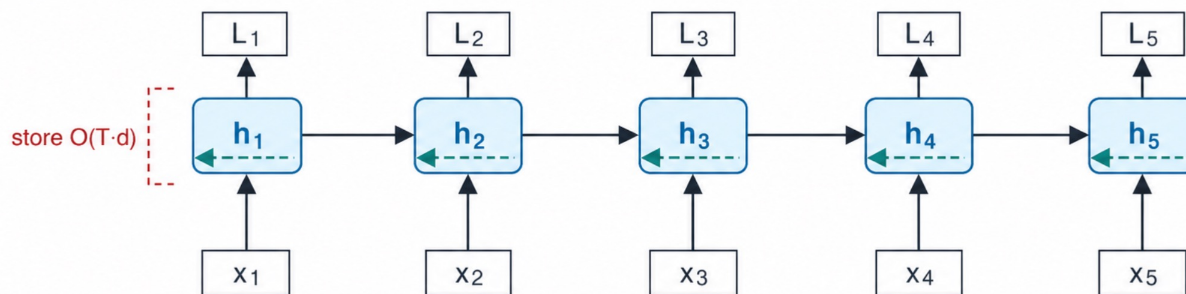
gradient diverges; weights shoot off to NaN

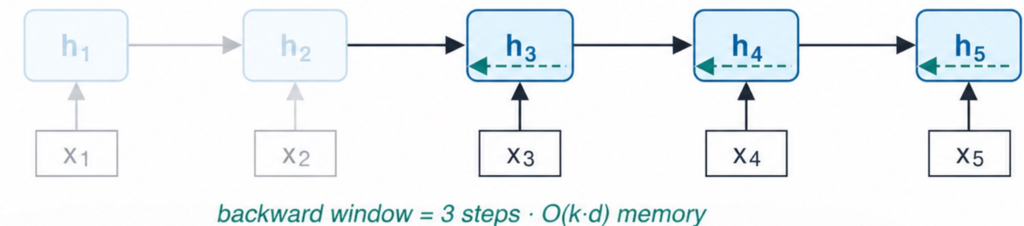
Backpropagation Through Time (BPTT)

Once unrolled, an RNN is just a deep feed-forward net with depth = sequence length and tied weights. Standard backprop works — but with two costs:

- **Memory:** must store every h_t for the backward pass — O(T·d).
- **Gradient through W_hh:** the chain rule multiplies T Jacobian factors. This is where vanishing/exploding lives.

Engineering fix — truncated BPTT: backprop only over the last k steps (typical k = 32, 64, 128). Trades long-range learning for tractable memory; standard in practice for T > 1000.





Cheap hacks help — but don't fix the structure

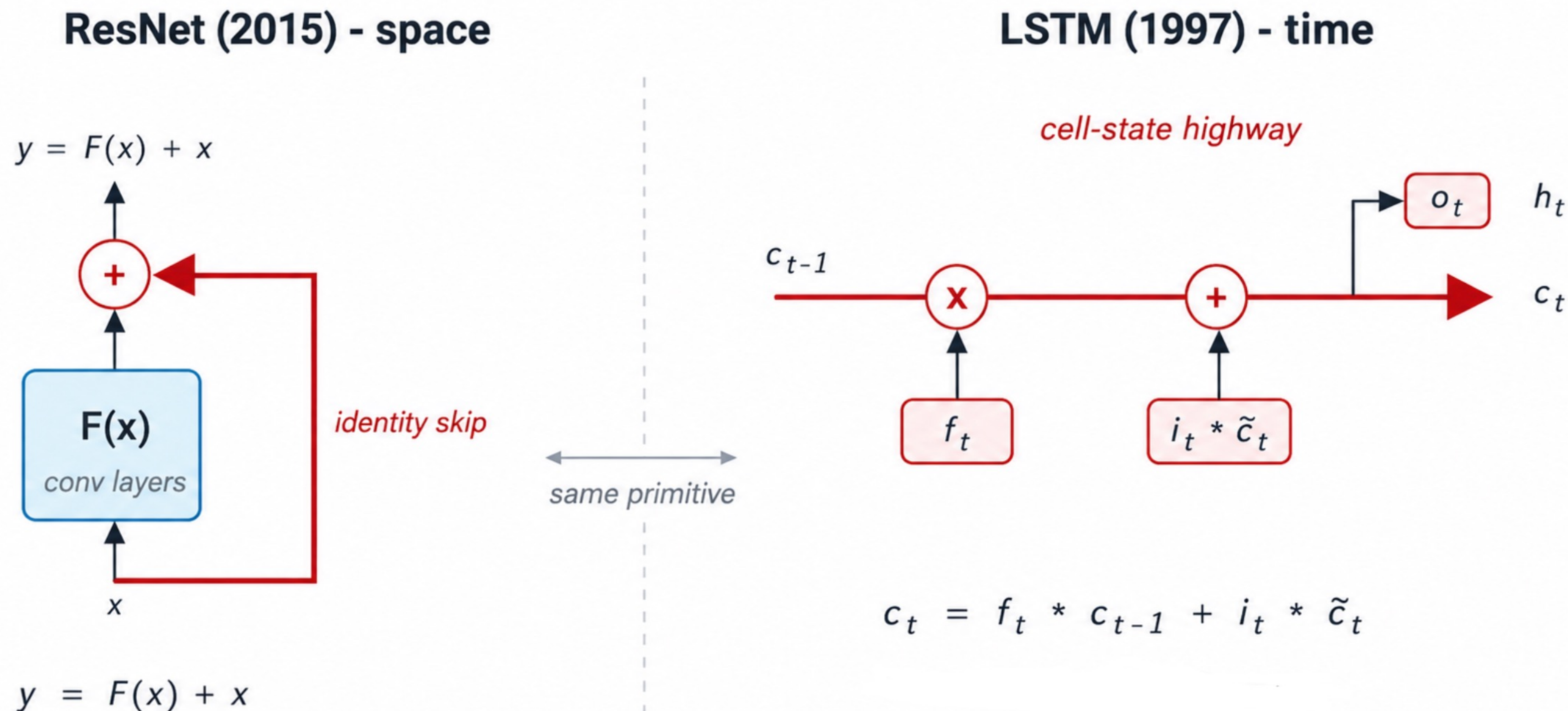
- **Gradient clipping (Pascanu et al., 2013):** if $\|\nabla\| > \tau$, rescale to τ. Cures exploding only.
- **ReLU activations:** linear in the positive regime — but no upper bound makes explosion worse without clipping.
- **Orthogonal initialization (Saxe et al., 2014):** initialize W_{hh} with $\lambda_{max} = 1$. Helps at step 0 but drifts during training.
- **Smaller learning rate, careful init, layer norm:** all extend the runway; none change the asymptote.

None of these change the multiplicative chain. We need a different architecture — one with an **additive identity path**.

3. The fix: gating & identity paths

What if the recurrent map were the identity?

- This is the ResNet (2015) insight. 18 years earlier: it is the LSTM insight too.
- Same primitive, two domains.
  - ResNet: identity skips through depth;
  - LSTM: an identity path through time.



The LSTM cell: a regulated additive highway

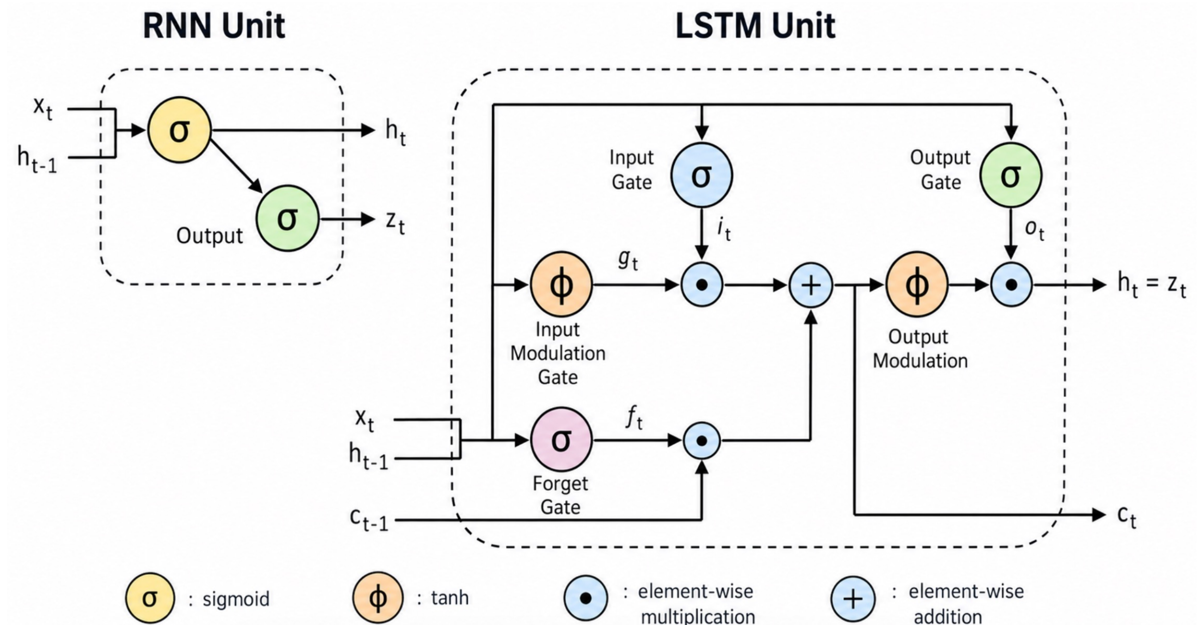
Hochreiter & Schmidhuber (1997).

Two states: h_t (output) and c_t (long-term memory).

Three gates regulate three operations:

- forget f_t — multiplicatively filters old memory
- input i_t — controls how much of the new candidate is written
- output o_t — controls how much memory is exposed to the rest of the network

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

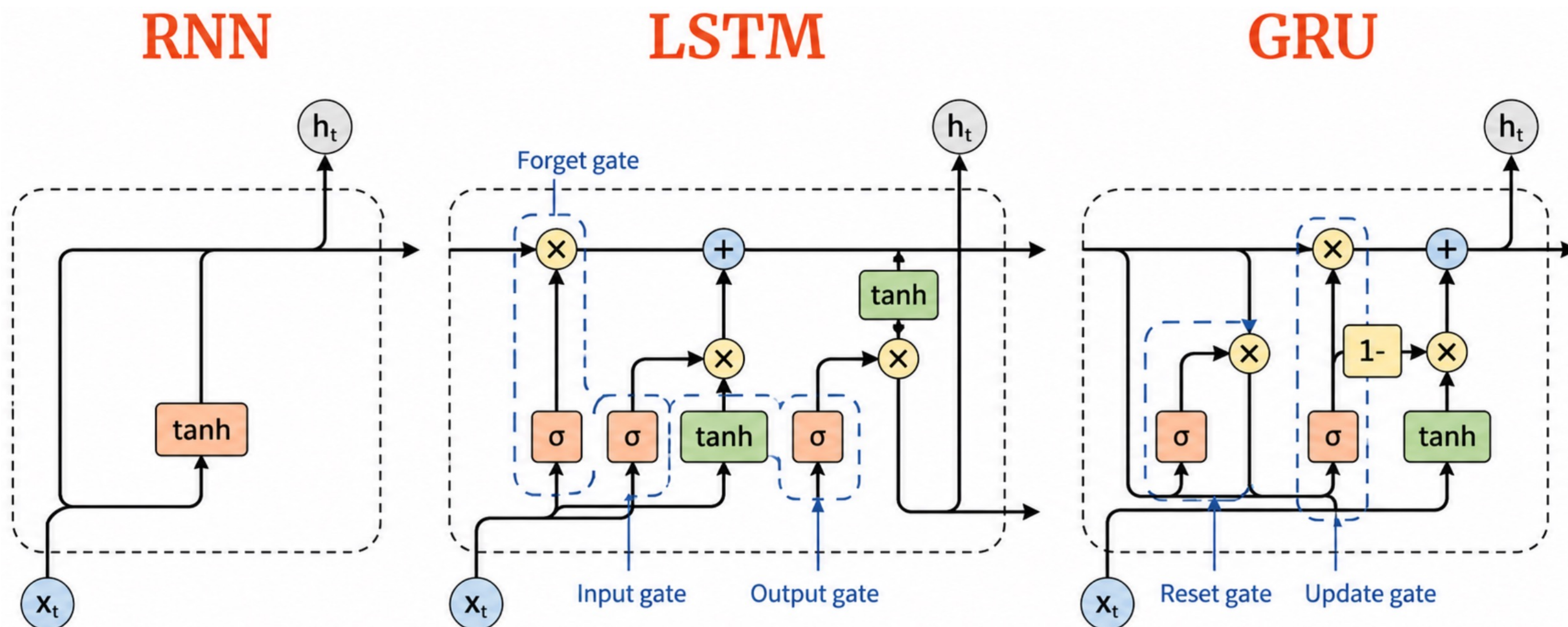


GRU — simpler, often just as good

Cho et al. (2014). Two key simplifications relative to LSTM:

- merge cell state and hidden state into one — h_t carries everything
- merge forget and input into one update gate z_t, with the constraint forget = 1 − input

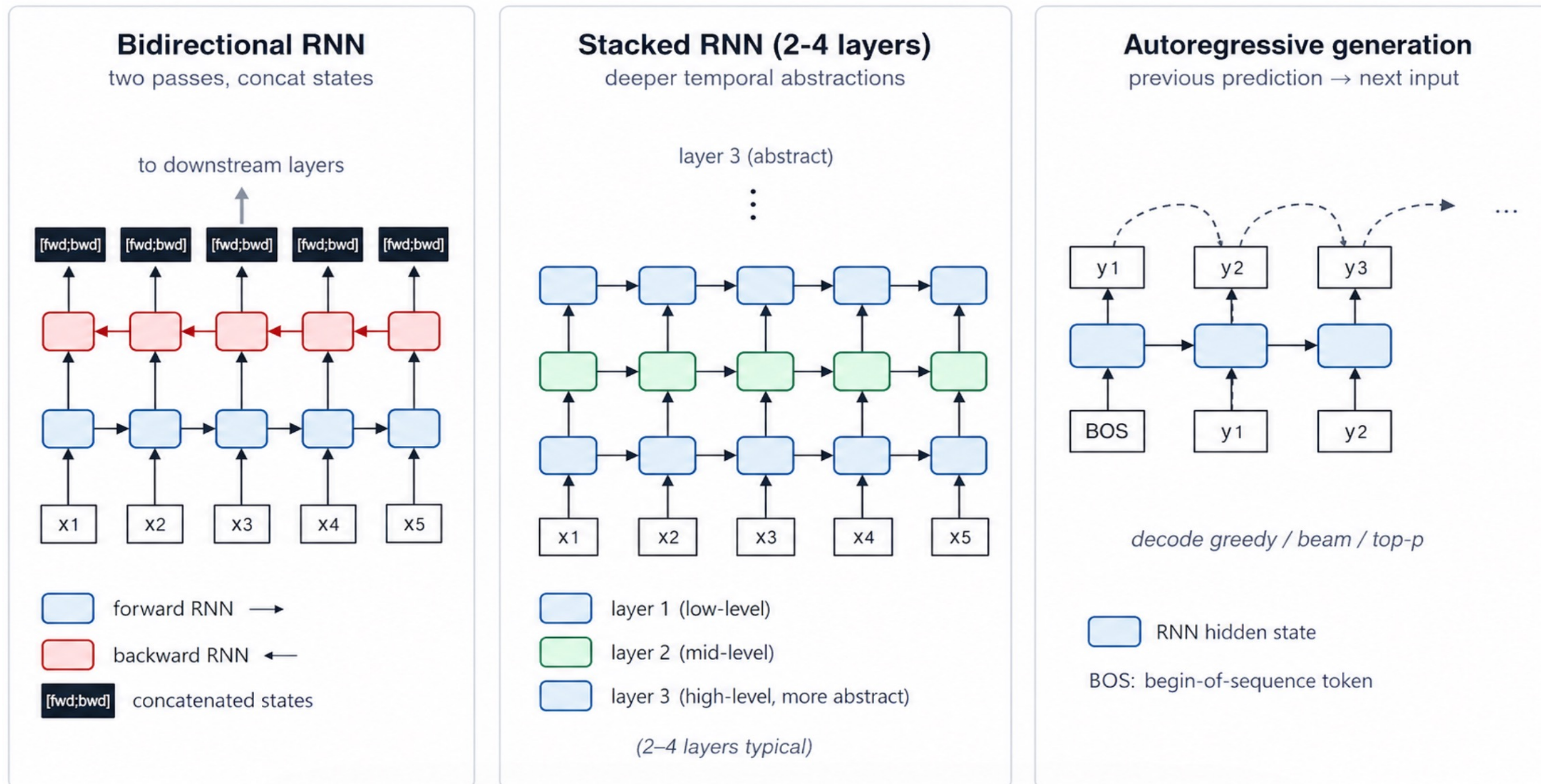
Result: ~25% fewer parameters, comparable accuracy on most tasks (Chung et al., 2014; Greff et al., 2017).



4. Building with RNNs

Three composition patterns

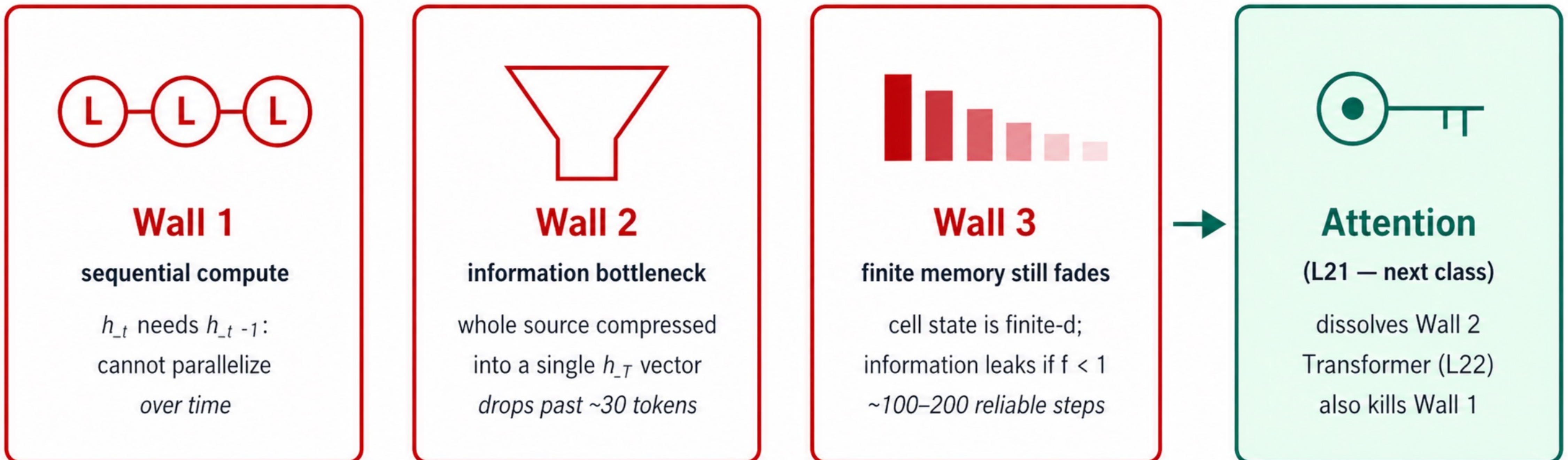
- Bidirectional: Two passes, concatenate states. Cheap, almost always helps.
- Stacked: depth in the recurrence. Higher layers see more abstract patterns.
- Autoregressive generation: feed the predicted $\hat{y}_t$ back as next input.



5. What RNNs can't fix

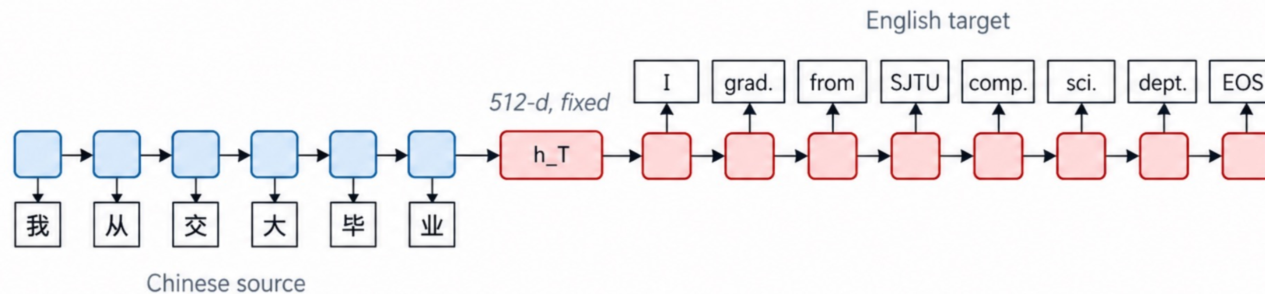
Three walls that gating cannot break

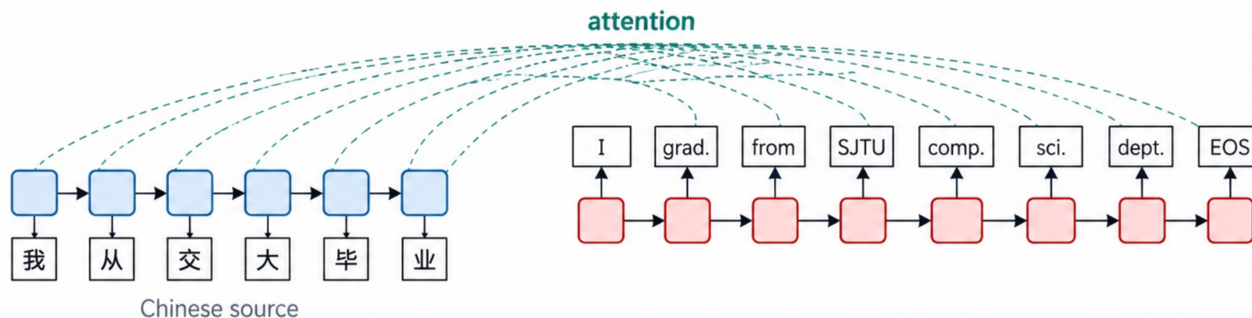
- LSTMs solved the gradient problem. But three other walls remained — and motivate everything in L21–L22.



Wall 2 in detail — the seq2seq bottleneck

- In encoder–decoder translation (slide 9, panel 4), the entire source sentence is compressed into a single h_T.
- The fix defined the next decade: let the decoder, at each step, look back and attend to all encoder states — not just the last one (next lecture).





Foreshadowing — what attention will give us

Three properties simultaneously, none of which an LSTM has:

- **Direct access.** Any output position can read any input position in O(1) hops. Wall 2 (bottleneck) gone.
- **Parallelizable.** All positions computed simultaneously — at training time. Wall 1 (sequential compute) gone, at least for training.
- **Content-based addressing.** Which position to attend to is learned, not fixed. More expressive than convolution's spatial inductive bias.

The cost: $O(T^2)$ memory and compute, vs RNN's $O(T)$.

Summary

- **One cell.** Recurrence is the sequence analogue of convolution: a shared, local operator applied across an axis (time, not space).
- **One trick.** The additive identity path — "skip connections through time" — is what makes deep recurrence trainable.
- **Three walls.** Sequential compute, single-vector bottleneck, finite fading memory.

Next lecture (L21) — Attention

How to let every output position look at every input position, learnably and efficiently?

Three things you should pre-load in your head:

- the seq2seq encoder–decoder picture
- dot products as similarity
- softmax as a soft argmax