Sequence Prediction and Representation (continued)

ECE 57000 — October 26, 2026

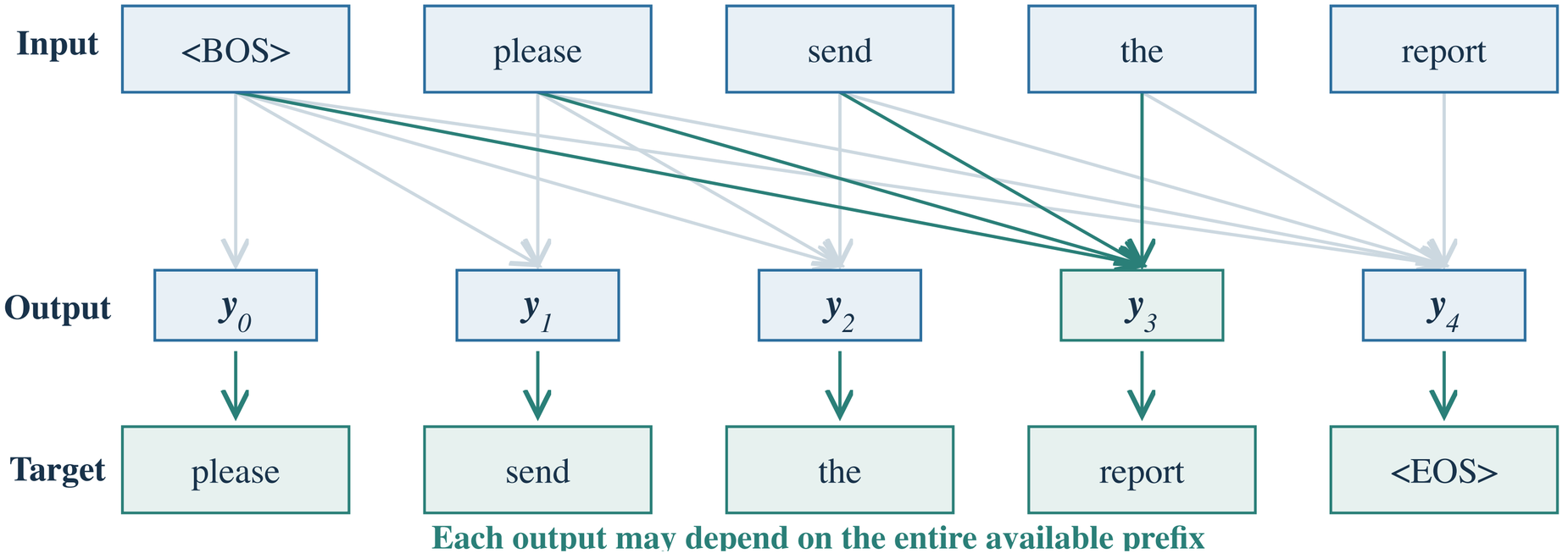
David I. Inouye

Next-token prediction connects text, learned features and targets

- Classification summarizes an image; segmentation preserves a prediction at every location.
- Token IDs select rows of a learned embedding table; ID differences do not measure meaning.
- Each prefix predicts the following token, using only the context available at that position.

Today: finish the training examples, then compare fixed windows, recurrent memory and attention.

Each input position predicts the token immediately after it



Each output may depend on the entire available prefix

$$\boldsymbol{p}_\ell = \mathrm{softmax}(\boldsymbol{y}_\ell), \qquad \mathcal{L}_\ell = -\log p_{\ell, w_{\ell+1}}$$

Output y_ℓ may depend on $w_0, \ldots, w_\ell$: all previous tokens **and the current token**.

The sequence model preserves positions while changing features

Stage	Tensor shape	Meaning
Token IDs	`[B,L]`	Discrete input tokens
Input sequence X	`[B,L,d]`	Embedded rows $\boldsymbol{x}_\ell$
Hidden sequence H	`[B,L,h]`	Hidden-state rows $\boldsymbol{h}_\ell$
Output sequence Y	`[B,L,V]`	Logit rows $\boldsymbol{y}_\ell$
Target IDs	`[B,L]`	Next-token labels

For one sequence: $\boldsymbol{x}_\ell = X[\ell, :], \boldsymbol{h}_\ell = H[\ell, :], \boldsymbol{y}_\ell = Y[\ell, :]$ (**rows**).

Padding is absent data: exclude padded targets from the loss.

Test completion on held-out email threads

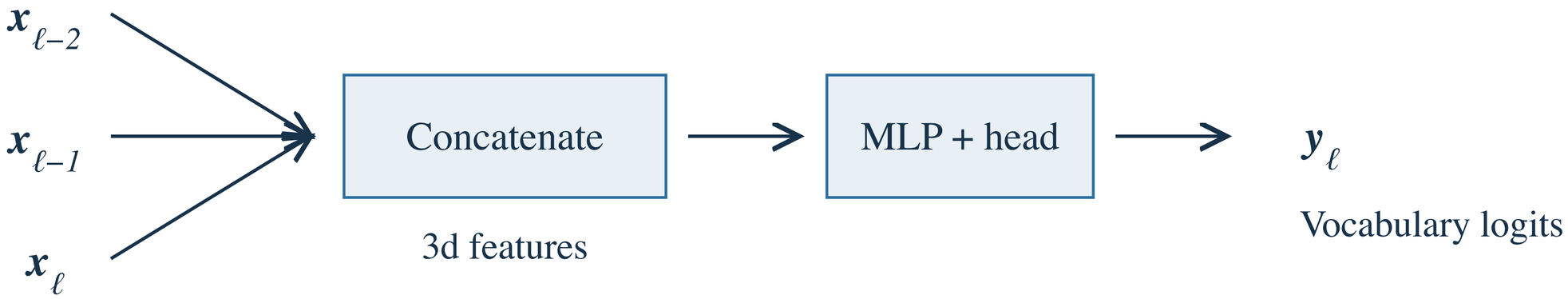
Group and split threads	Then create examples
Training threads	Training prefixes and next-token targets
Validation threads	Validation prefixes and next-token targets
Reserved test threads	Final held-out prefixes and targets

- Overlapping windows from one email share much of their text.
- Fit the vocabulary on **training emails only**; handle unseen tokens explicitly.

Now build a baseline: how much of the prefix should it read?

A fixed window turns continuation into familiar supervised prediction

Last three token embeddings



Earlier tokens are outside this predictor's input.

$$\boldsymbol{y}_\ell = f_\theta([\boldsymbol{x}_{\ell-2}\ \boldsymbol{x}_{\ell-1}\ \boldsymbol{x}_\ell]) \in \mathbb{R}^v$$

Its limitation: anything before the window is unavailable to the model.

Which change lets the model distinguish these contexts?

“Send the **report** today. Please attach the …”

“Send the **slides** today. Please attach the …”

> **Which change permits different predictions from a three-token model?**
> **A. Add more hidden units**
> **B. Include earlier tokens**
> **C. Train on more emails**

B. Identical input windows produce the same predicted distribution for fixed parameters.

A recurrent state carries earlier information into the next step

Fixed-window predictor	Recurrent predictor
Reads the most recent k tokens	Reads the current token and previous state
Earlier tokens fall outside the input	Earlier tokens can affect the state
Input width depends on k	State width stays fixed

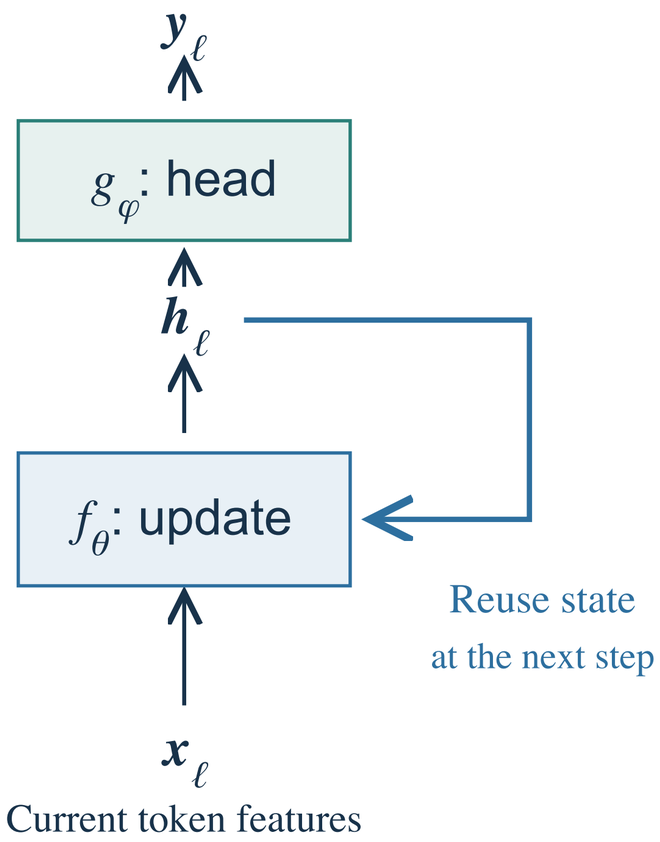
Next question: how can one learned update carry useful earlier information forward?

Recurrent Neural Networks (RNN)

David I. Inouye

Dynamic Content · Last Updated: October 26, 2026

One transition combines the current token and the previous state



State update

$$\boldsymbol{h}_\ell = f_\theta(\boldsymbol{x}_\ell, \boldsymbol{h}_{\ell-1})$$

Vocabulary head

$$\boldsymbol{y}_\ell = g_\phi(\boldsymbol{h}_\ell)$$

The state changes with the prefix. The parameters θ, ϕ are shared.

Next: choose a concrete f_θ and g_ϕ.

Familiar layers implement the recurrent update

$$\boldsymbol{h}_\ell = \mathrm{ReLU}(\boldsymbol{x}_\ell W_x + \boldsymbol{h}_{\ell-1} W_h + \boldsymbol{b})$$

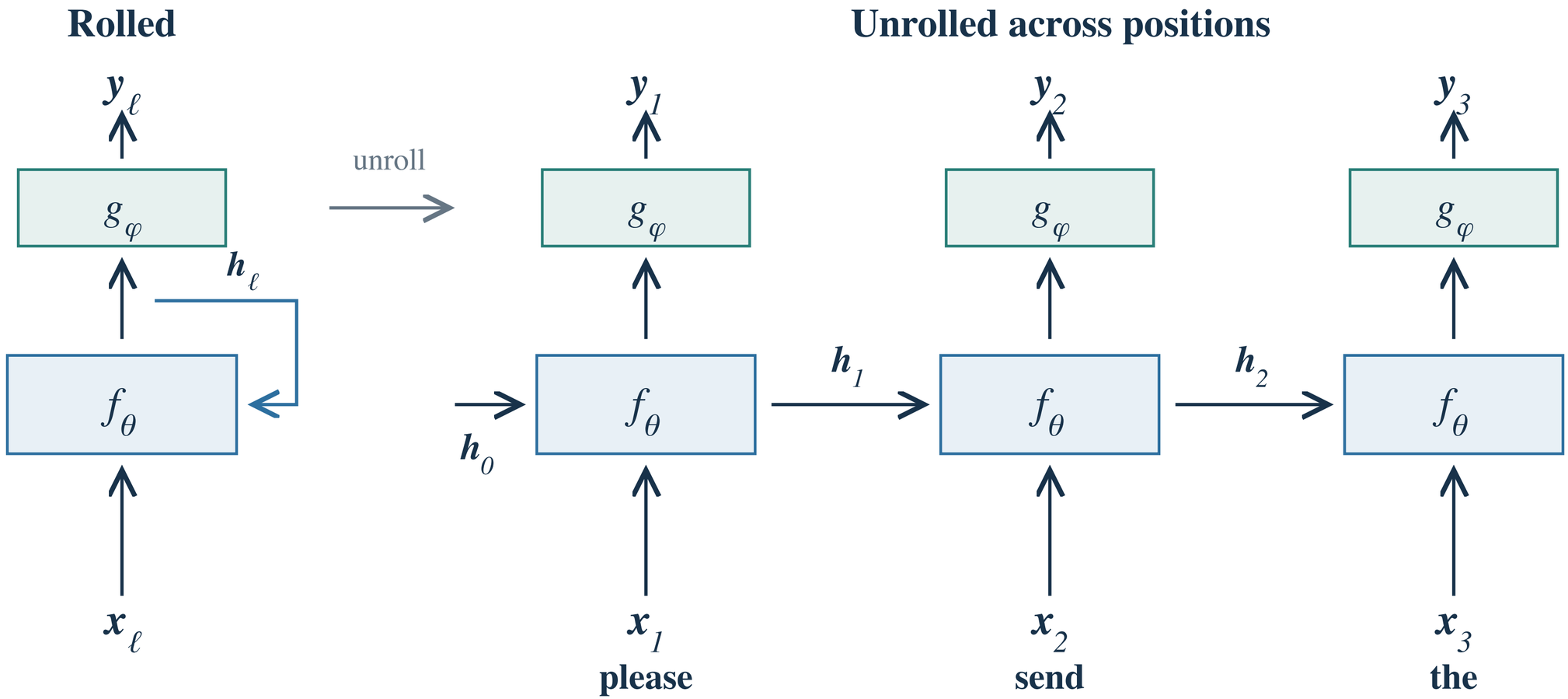
$$\boldsymbol{y}_\ell = \boldsymbol{h}_\ell W_y + \boldsymbol{c}, \qquad \boldsymbol{p}_\ell = \mathrm{softmax}(\boldsymbol{y}_\ell)$$

Embedding and state	Learned matrices	Vocabulary output
$\boldsymbol{x}_\ell \in \mathbb{R}^d$	$W_x \in \mathbb{R}^{d \times h}$	$\boldsymbol{y}_\ell \in \mathbb{R}^v$
$\boldsymbol{h}_\ell \in \mathbb{R}^h$	$W_h \in \mathbb{R}^{h \times h}$	$W_y \in \mathbb{R}^{h \times v}$

$f_\theta \colon \theta = (W_x, W_h, \boldsymbol{b})$. $g_\phi \colon \phi = (W_y, \boldsymbol{c})$.

Unroll these shared functions to see the computation across the sequence.

Unrolling repeats computation with shared parameters



Three drawn cells are **three uses of one learned transition**.

A three-token trace separates input, state and prediction

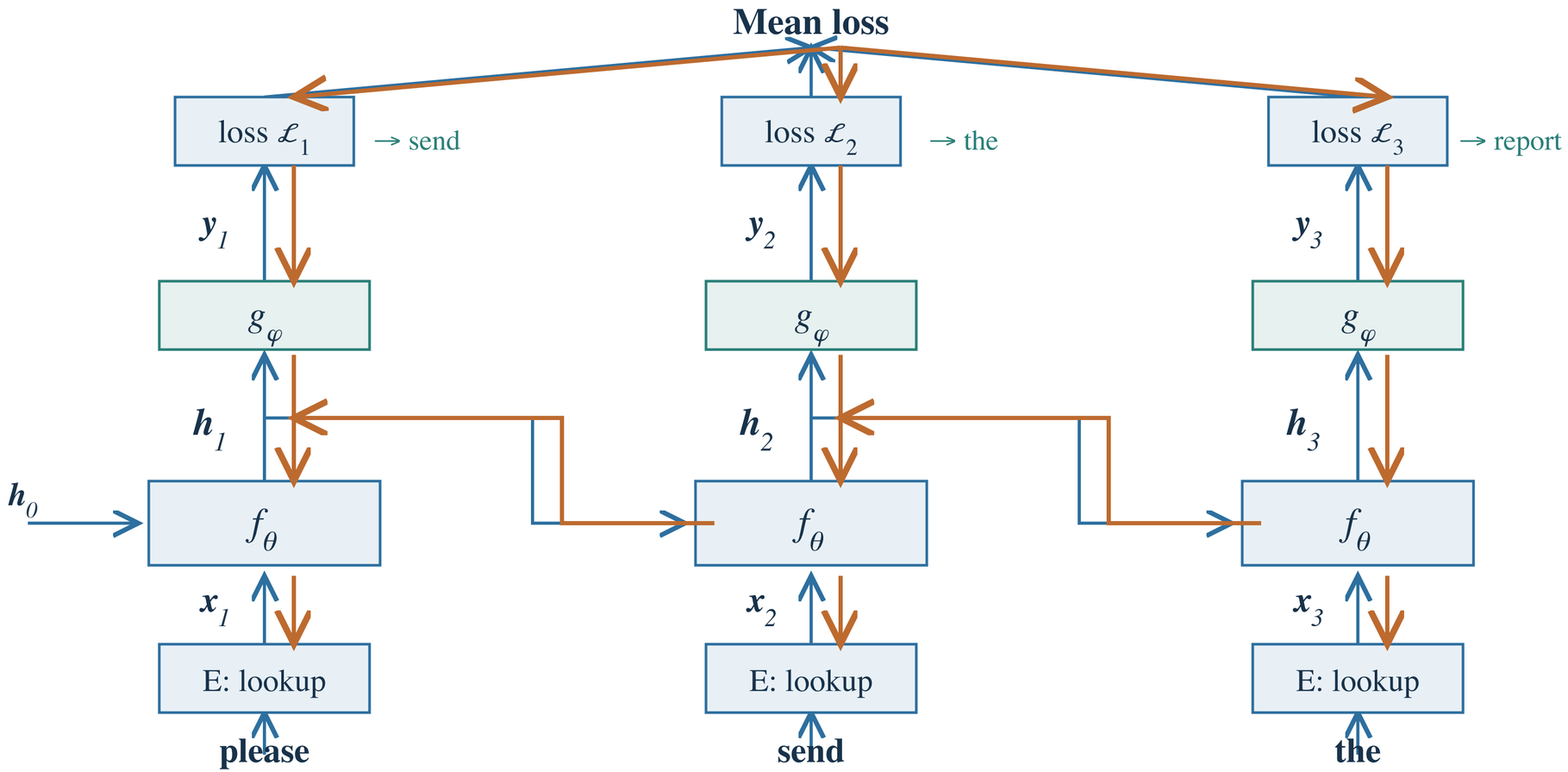
Synthetic parameters: $W_x = I_2, W_h = 0.5I_2, \boldsymbol{b} = \mathbf{0}, \boldsymbol{h}_0 = \mathbf{0}$.

Token	Embedding $\boldsymbol{x}_\ell$	Updated state $\boldsymbol{h}_\ell = \mathrm{ReLU}(\boldsymbol{x}_\ell + 0.5\boldsymbol{h}_{\ell-1})$
`please`	$[1, 0]$	$[1, 0]$
`send`	$[0, 1]$	$[0.5, 1]$
`the`	$[1, 1]$	$[1.25, 1.5]$

A head column $[1, 1]^\mathsf{T}$ gives the `report` logit 2.75 at the last step.

That logit is a score; softmax still compares it with **all vocabulary logits**.

Training assigns credit through the unrolled computation



Accumulate gradients for shared θ, ϕ and E; update each parameter once

Blue: forward values • Orange: partial derivatives flowing backward

Repeated derivatives can weaken or amplify distant credit

Scalar illustration: $h_\ell = \mathrm{ReLU}(a h_{\ell-1} + x_\ell)$, with all relevant preactivations positive.

$$\frac{\partial h_\ell}{\partial h_{\ell-1}} = a, \qquad \frac{\partial h_\ell}{\partial h_{\ell-n}} = \prod_{j=\ell-n+1}^{\ell} \frac{\partial h_j}{\partial h_{j-1}} = a^n$$

Local derivative	10 steps	20 steps	100 steps
$a = 0.5$	9.77×10^{-4}	9.54×10^{-7}	7.89×10^{-31}
$a = 1.5$	57.7	3.33×10^{3}	4.07×10^{17}

A longer path can make earlier information difficult to learn to preserve.

A longer message reuses the same learned transition

An RNN was trained on 20-token messages and receives a 30-token message.

> **What must grow to process the longer message?**
> **A. The number of learned transition matrices**
> **B. The number of recurrent computation steps**
> **C. The width of each hidden state**

B. More steps reuse the same parameters and state width.

A GRU learns what to retain and what to replace

1 · Gates

$$\boldsymbol{u}_\ell = [\boldsymbol{h}_{\ell-1}, \boldsymbol{x}_\ell]$$

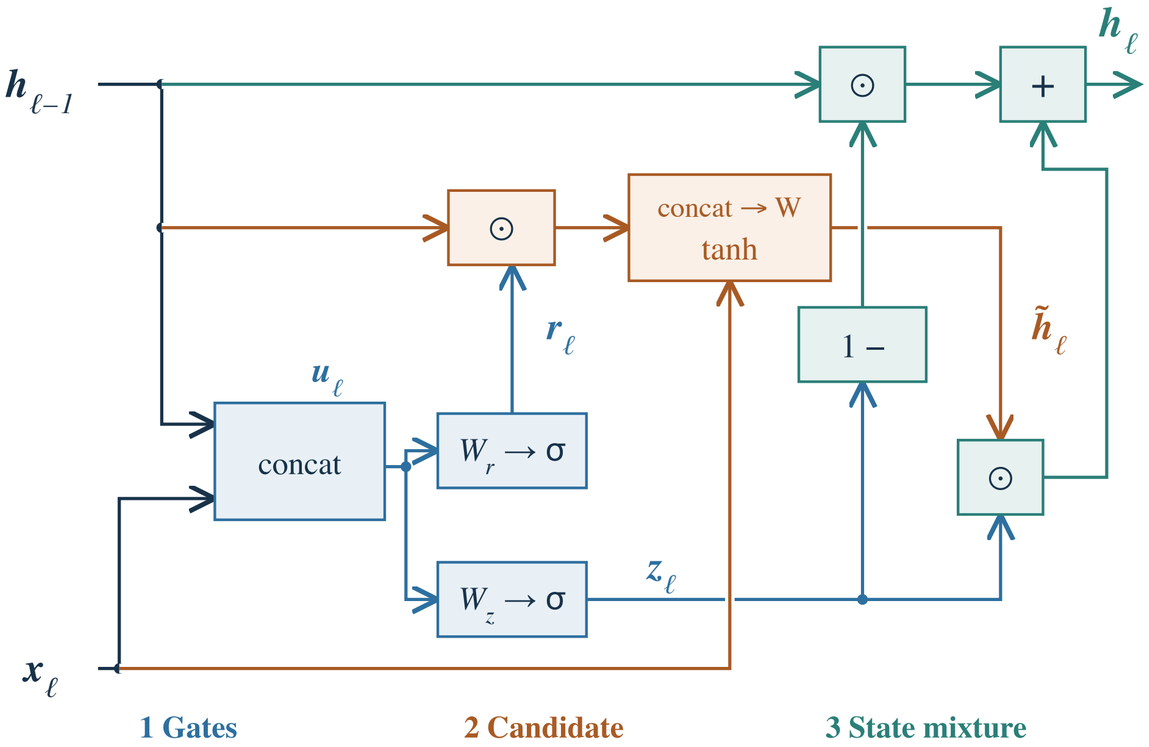
$$\boldsymbol{r}_\ell = \sigma(\boldsymbol{u}_\ell W_r), \quad \boldsymbol{z}_\ell = \sigma(\boldsymbol{u}_\ell W_z)$$

2 · Candidate

$$\widetilde{\boldsymbol{h}}_\ell = \tanh([\boldsymbol{r}_\ell \odot \boldsymbol{h}_{\ell-1}, \boldsymbol{x}_\ell]W)$$

3 · New recurrent state

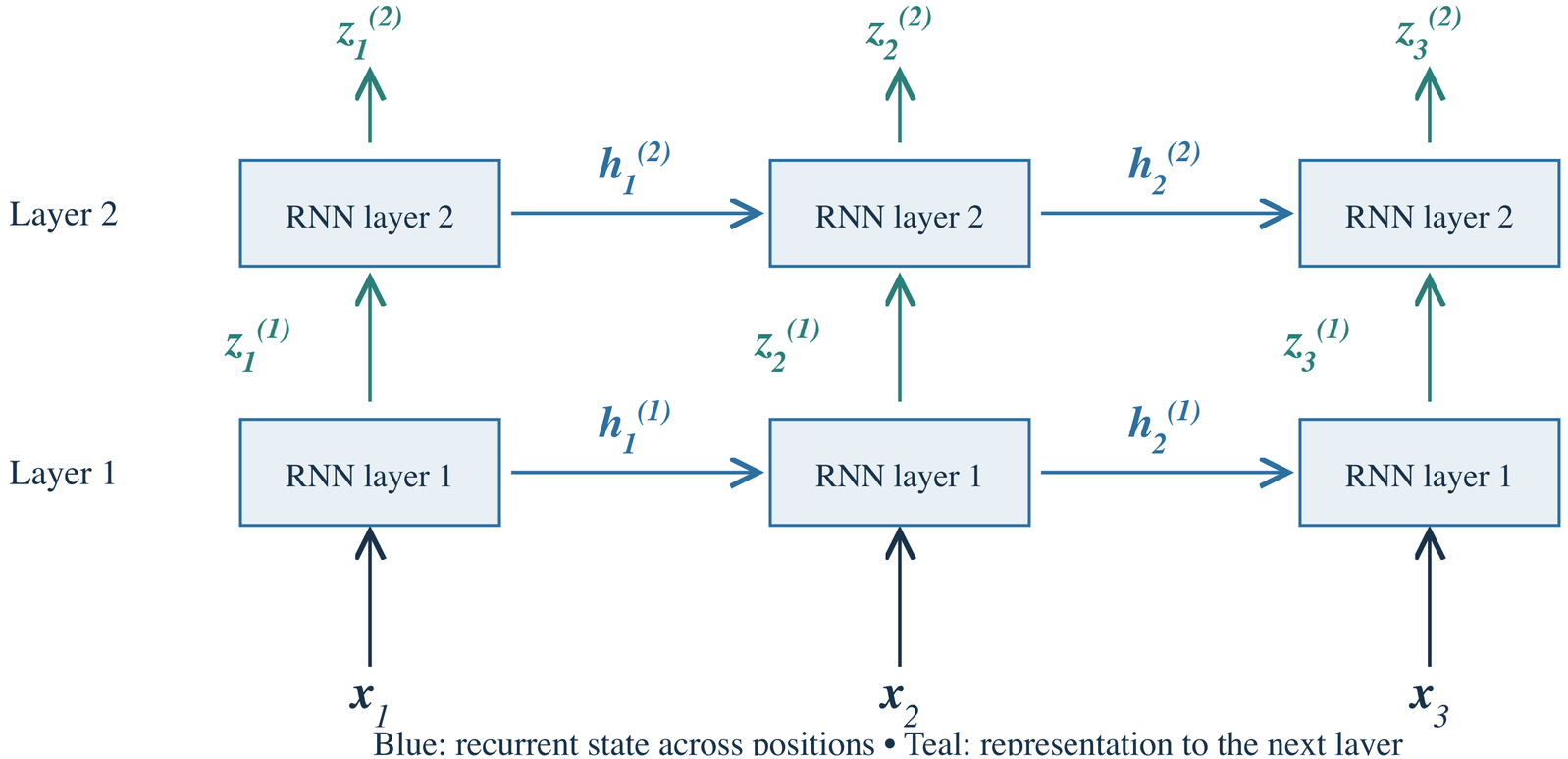
$$\boldsymbol{h}_\ell = (1 - \boldsymbol{z}_\ell) \odot \boldsymbol{h}_{\ell-1} + \boldsymbol{z}_\ell \odot \widetilde{\boldsymbol{h}}_\ell$$



$\boldsymbol{z}_\ell \approx \mathbf{0}$: retain old state. $\boldsymbol{z}_\ell \approx \mathbf{1}$: use candidate.

LSTM: a common, more complex gated alternative with a separate cell state.

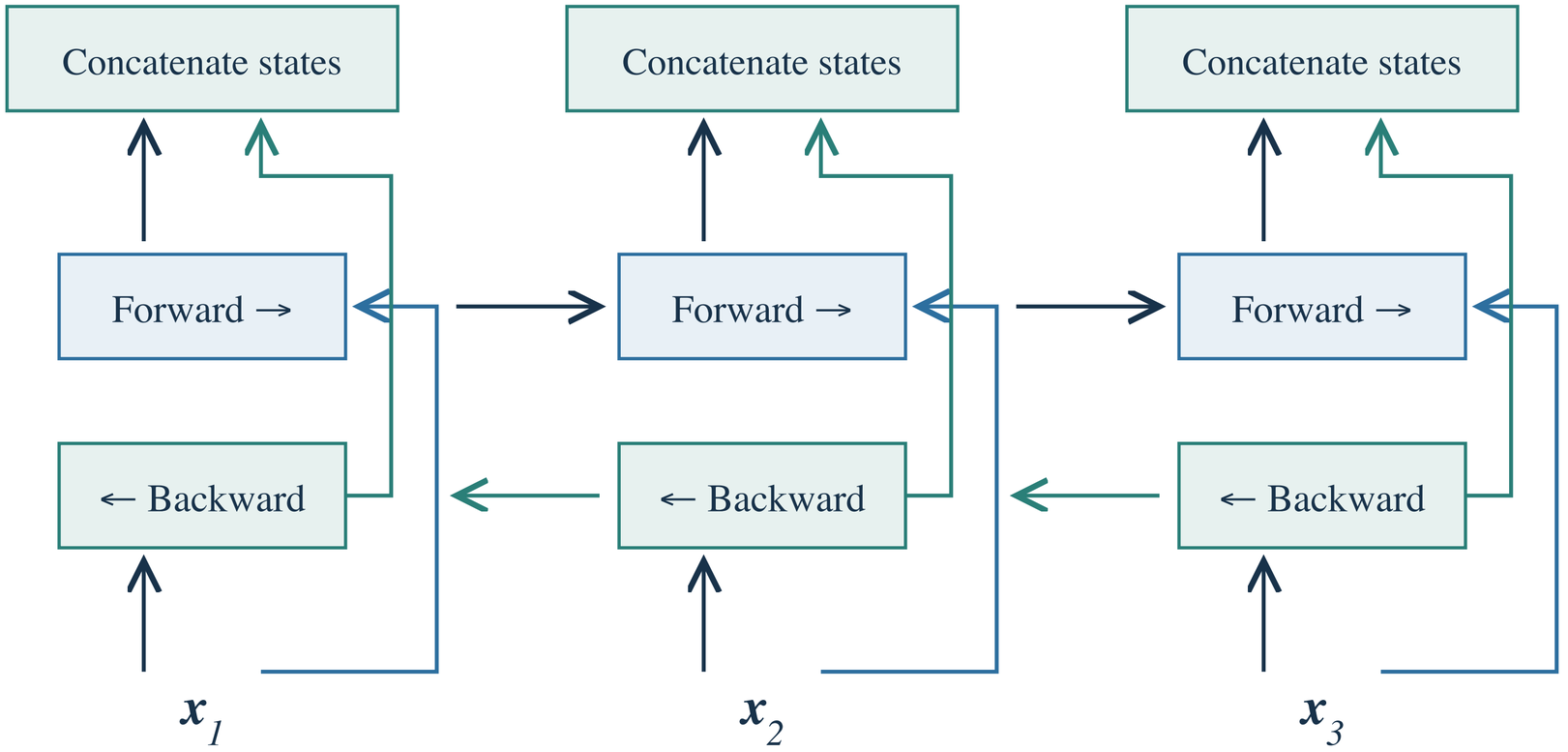
Deep RNNs stack recurrent layers



Horizontal: $h_\ell^{(k)}$, recurrent state. **Vertical:** $z_\ell^{(k)}$, layer representation.

For vanilla RNNs and GRUs, $z_\ell^{(k)} = h_\ell^{(k)}$: same value, different role.

Bidirectional RNNs combine left and right context



Useful when the **whole input is available**, such as tagging a complete sentence.

Not valid for next-token prediction if the backward state can see the target.

PyTorch runs the recurrence; `backward()` performs BPTT

`ids` and shifted `targets`: `[B,L]`. Each row is a separate email.

```
model = nn.ModuleDict({
    "embed": nn.Embedding(V, 64, padding_idx=PAD),
    "rnn": nn.RNN(64, 96, nonlinearity="relu", batch_first=True),
    "head": nn.Linear(96, V),
})
opt = torch.optim.Adam(model.parameters(), lr=0.002)

# Repeat for each training batch:
opt.zero_grad(set_to_none=True)
X = model["embed"](ids)                      # [B,L,64]
H, _ = model["rnn"](X)                       # [B,L,96]
Y = model["head"](H)                         # [B,L,V] logits
loss = F.cross_entropy(Y.transpose(1, 2), targets, ignore_index=PAD)
loss.backward()                             # BPTT: head → RNN → embeddings
nn.utils.clip_grad_norm_(model.parameters(), 1.0)
opt.step()                                  # one shared-parameter update
```

Forward builds the graph. Backward accumulates gradients. The optimizer updates weights.

A small RNN improves held-out email next-token prediction

Measured AESLC email-body experiment: 13,415 train / 1,769 validation / 1,604 test emails.

Model	Test log loss ↓ (nats/token)	Exact next-token accuracy ↑
Bigram: use only the previous token	4.840	18.9%
One-layer RNN: use the prefix	**4.454**	**23.3%**

- At most **64 tokens/email**; shared **4,000-token vocabulary**; one training seed.
- **6.5% of test targets become** `<UNK>`; accuracy on known targets is 17.3% vs. 21.8%.

Synthetic prompt, measured predictions: `please send the` → **attached**, **following**, **latest**.

Next comparison: freeze these data, tokens and metrics; replace the RNN with attention.

Data: Zhang & Tetreault, AESLC (2019), CC BY-NC-SA 4.0. Local run, October 26, 2026.

Recurrent memory creates information and training bottlenecks

Limitation	Why it matters
Fixed-width memory	Earlier information must survive repeated state updates; a final-state summary must represent the whole sequence.
Long gradient paths	Distant credit passes through many Jacobians, complicating optimization.
Sequential computation	State $\boldsymbol{h}_\ell$ must wait for $\boldsymbol{h}_{\ell-1}$, limiting training parallelism across positions.

Next: retain all position representations and combine them directly.