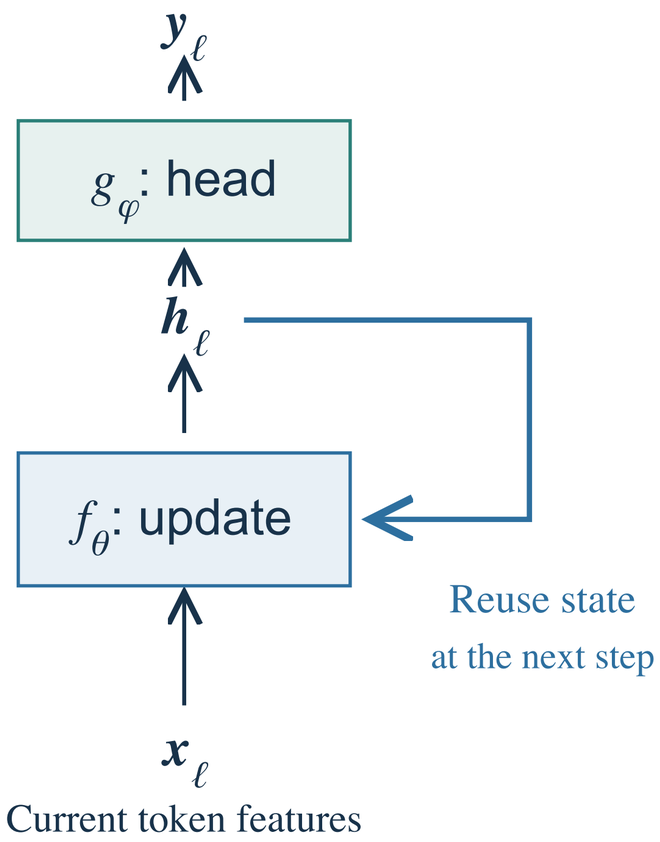
Recurrent Neural Networks (RNN)

David I. Inouye

Dynamic Content · Last Updated: October 30, 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

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

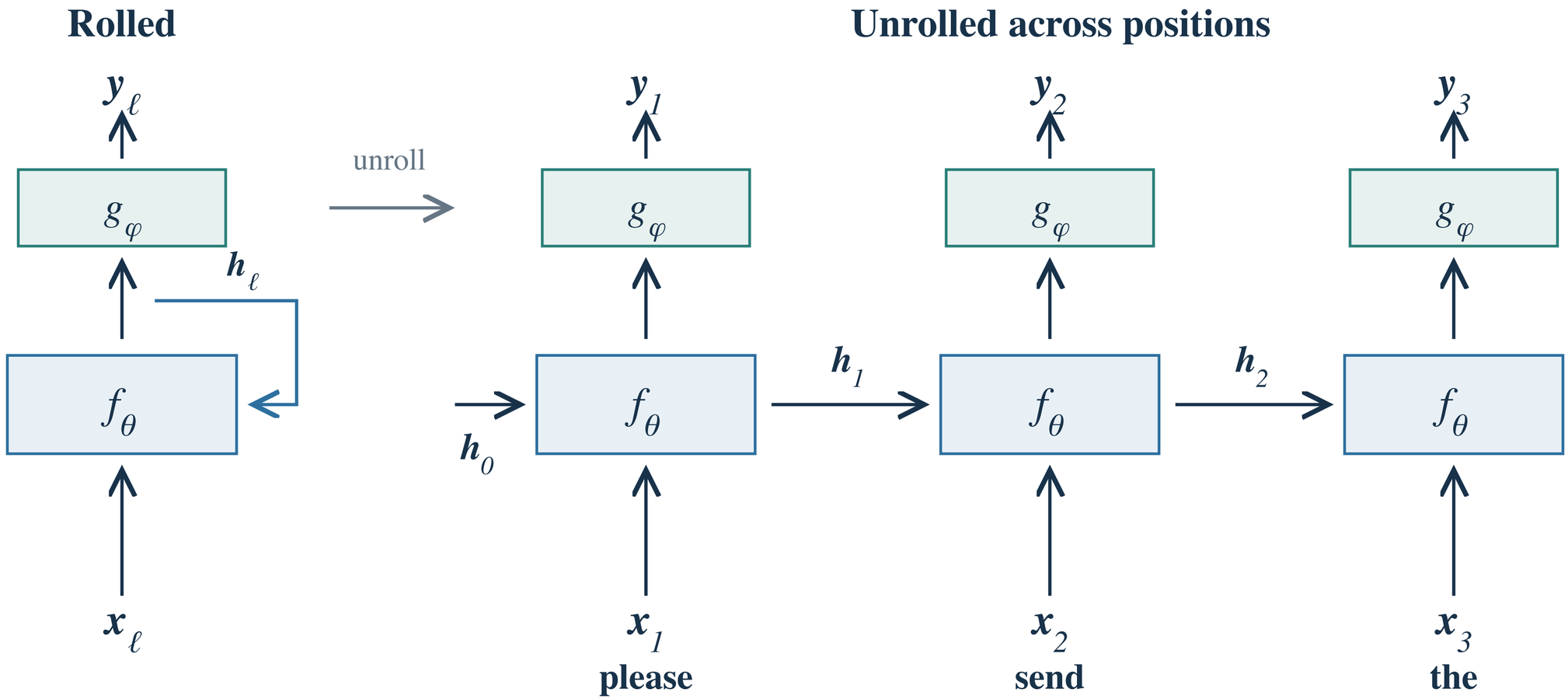
$$\boldsymbol{y}_\ell = W_y \boldsymbol{h}_\ell + \boldsymbol{c}, \qquad \boldsymbol{p}_\ell = \mathrm{softmax}(\boldsymbol{y}_\ell)$$

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

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

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

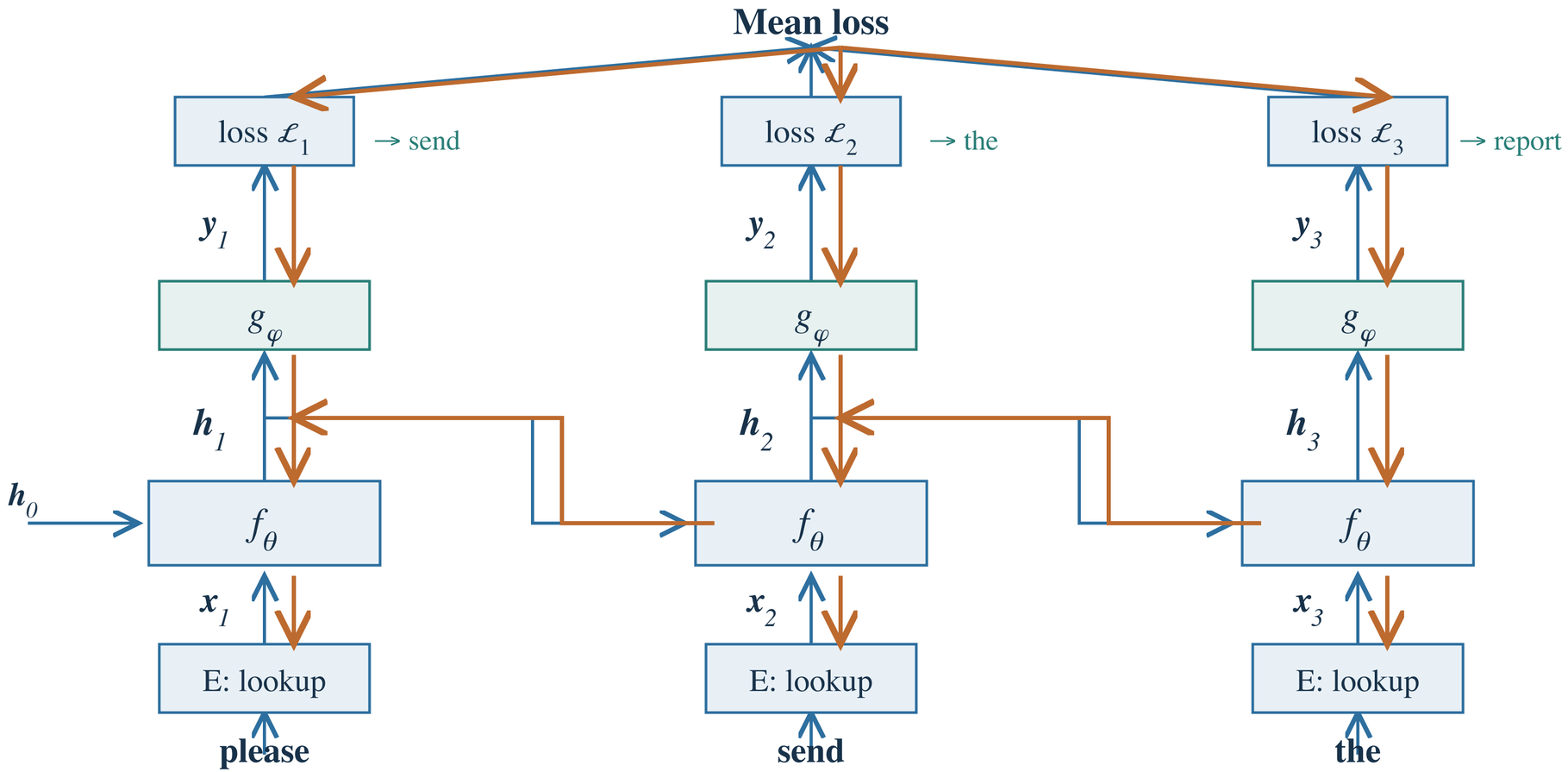
Synthetic parameters: $W_x = I_2, W_h = 0.5I_2, \boldsymbol{b} = \mathbf{0}, \boldsymbol{h}_0 = \mathbf{0}$.

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

A head row $[1, 1]$ 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

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

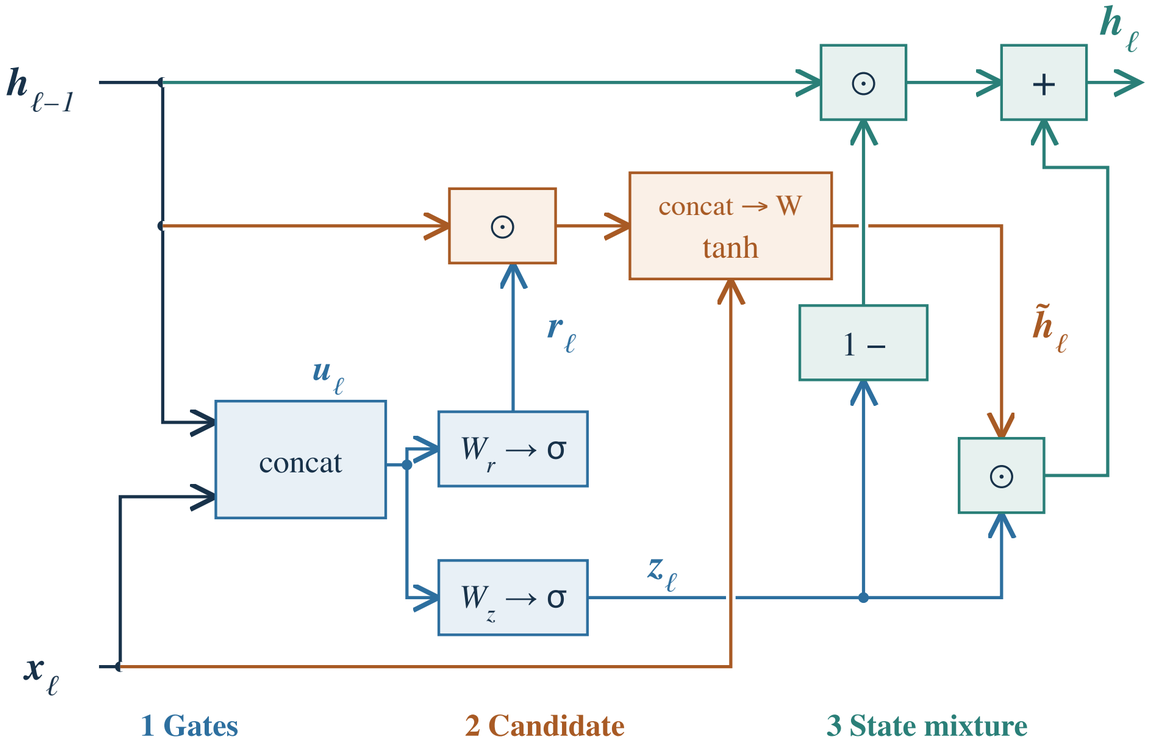
$$\boldsymbol{r}_\ell = \sigma(W_r \boldsymbol{u}_\ell), \quad \boldsymbol{z}_\ell = \sigma(W_z \boldsymbol{u}_\ell)$$

2 · Candidate

$$\widetilde{\boldsymbol{h}}_\ell = \tanh\left(W \begin{bmatrix} \boldsymbol{r}_\ell \odot \boldsymbol{h}_{\ell-1} \\ \boldsymbol{x}_\ell \end{bmatrix}\right)$$

3 · New recurrent state

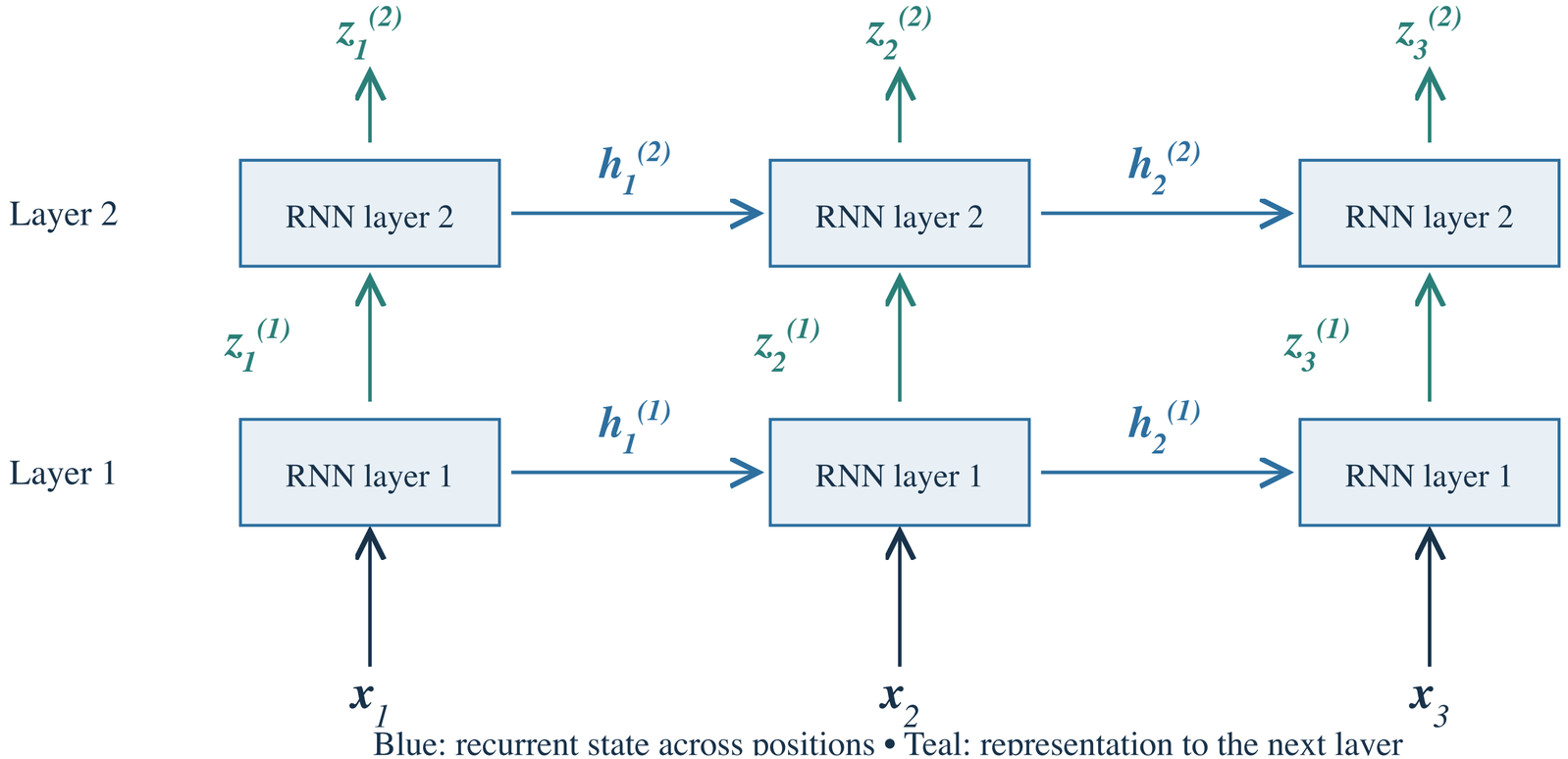
$$\boldsymbol{h}_\ell = (1 - \boldsymbol{z}_\ell) \odot \boldsymbol{h}_{\ell-1} + \boldsymbol{z}_\ell \odot \widetilde{\boldsymbol{h}}_\ell$$



$z_\ell \approx 0$: retain old state. $z_\ell \approx 1$: use candidate.

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

Deep RNNs stack recurrent layers

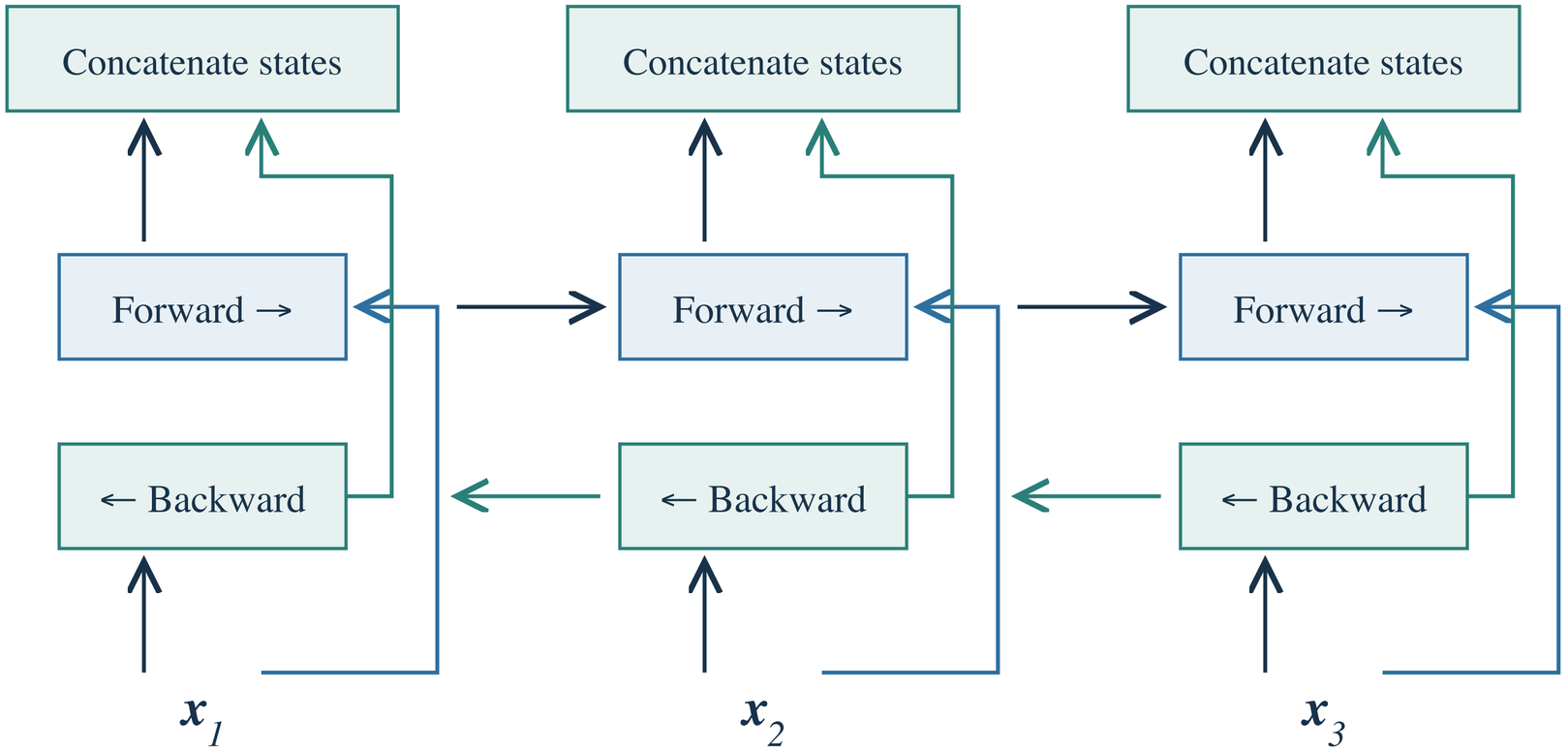


Blue: recurrent state across positions • Teal: representation to the next layer

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.