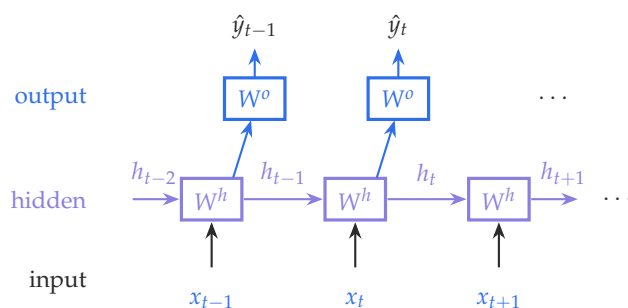


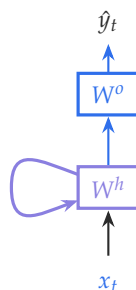
Recurrent Neural Networks 2

DS 4440 · LECTURE 12

1. Last time: RNNs for sequences



Or, folded up — the same cell applied again and again:

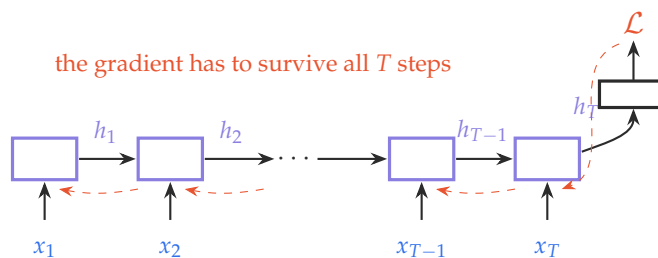


2. The problem: long-term dependencies

Suppose the label depends on the *first* symbol:

X	y
(0 0 0 0 0)	0
(1 0 0 0 0)	1
...	
(1 0 0 0 0)	1

To learn this, the signal must flow through the *entire* sequence — forward, so the answer is still available at the end, and backward, so the first step learns what to store.



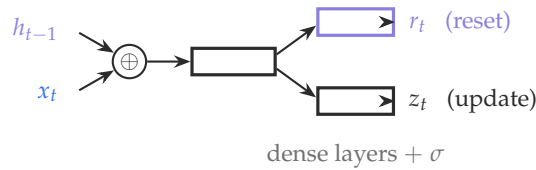
Last lecture we saw why it usually does not: each step multiplies by another Jacobian, and the product explodes or vanishes. What we want is a path along which the factor is *one* — a state the model can carry forward untouched.

3. Gated Recurrent Units (GRUs)

Similar to LSTMs (next) but simpler.

Idea. Introduce **gating mechanisms** that let the model update h — or opt not to, and skip.

Both gates are ordinary dense layers on $[h_{t-1}, x_t]$ followed by a sigmoid, so each gate is a vector of numbers in $(0, 1)$, one per hidden unit:



$$r_t = \sigma(x_t W_r^x + h_{t-1} W_r^h + b_r), \quad z_t = \sigma(x_t W_z^x + h_{t-1} W_z^h + b_z).$$

Reset: what the candidate is allowed to see

$$\tilde{h}_t \leftarrow \tanh(x_t W_h^x + \underbrace{(r_t \odot h_{t-1})}_{\text{permits forgetting}} W_h^h + b_h)$$

Where $r_t \approx 0$, the old state is zeroed out before it reaches the candidate — the model can start fresh.

Update: how much of the old state survives

$$h_t \leftarrow \underbrace{z_t \odot h_{t-1}}_{\text{copy}} + \underbrace{(1 - z_t) \odot \tilde{h}_t}_{\text{update}}$$

This is the piece that answers last lecture. If $z_t = 1$ then $h_t = h_{t-1}$ exactly — the state is copied, and a unit can hold a value across many steps by keeping its gate near 1.

Be careful about what that does to the gradient, though. Differentiating is not just a matter of reading off z_t : the term $z_t \odot h_{t-1}$ is a product of two things that *both* depend on h_{t-1} , since the gate is $z_t = \sigma(x_t W_z^x + h_{t-1} W_z^h + b_z)$. Applying the product rule to both terms of the update and collecting the two gate derivatives:

$$\frac{\partial h_t}{\partial h_{t-1}} = \underbrace{\text{diag}(z_t)}_{\text{copy path}} + \underbrace{\text{diag}(h_{t-1} - \tilde{h}_t)}_{\text{the gate itself moves}} \frac{\partial z_t}{\partial h_{t-1}} + \underbrace{(1 - z_t) \odot \frac{\partial \tilde{h}_t}{\partial h_{t-1}}}_{\text{through the candidate}}.$$

So there is still a Jacobian. What gating buys is the first term: a *near-identity* component that does not shrink with depth. The other two are suppressed exactly when the gate is doing its job — with z saturated near 1, σ' is tiny, which kills the middle term, and $(1 - z_t) \approx 0$ kills the last one. The gradient then travels back along something close to a straight copy, instead of being multiplied down at every step.

A trace: one unit remembering one bit

Take the task from §2 — the label is the *first* symbol — and set a GRU by hand with a single hidden unit, so every quantity below is a number rather than a vector. We want the unit to write x_1 into h at the first step and then hold it, no matter how long the sequence runs. Three weights do the work:

$$\underbrace{W_z^x = -10, W_z^h = +10, b_z = 0}_{\text{update gate}} \quad \underbrace{b_r = -10}_{\text{reset always closed}} \quad \underbrace{W_h^x = 3}_{\text{candidate}}$$

with the remaining weights zero and $h_0 = 0$. Read what each one does:

- $z_t = \sigma(-10 x_t + 10 h_{t-1})$. A 1 on the input drives z to 0 — *write*. A stored value with no input drives it to 1 — *copy*.
- $r_t = \sigma(-10) \approx 0$ always, so the candidate ignores the old state entirely and depends only on x_t .
- $\tilde{h}_t = \tanh(3x_t)$, which is 0.995 when $x_t = 1$ and 0 otherwise.

Trace it on $x = (1, 0, 0, 0, 0)$:

t	x_t	z_t	$\tilde{h}_t$	h_t	
0	—	—	—	0	
1	1	0.00005	0.995	0.9950	write
2	0	0.99995	0	0.9950	copy
3	0	0.99995	0	0.9949	copy
4	0	0.99995	0	0.9949	copy
5	0	0.99995	0	0.9948	copy

At $t = 1$ the gate is open: $h_1 = z \cdot 0 + (1 - z) \cdot 0.995 = 0.995$. After that the gate closes and each step is $h_t = 0.99995 h_{t-1}$ — a copy, to four decimal places. Run the same weights for 50 steps and h is still 0.993. A readout of $\sigma(10 h_T - 5)$ gives 0.993 for this sequence.

On $x = (0, 0, 0, 0, 0)$ the unit never writes anything: $\tilde{h} = 0$ at every step, so h stays at 0 whatever the gate does, and the readout gives 0.007. The two sequences differ only in their first symbol, and the model still tells them apart at step 5.

What a vanilla RNN does on the same task

Give a plain RNN, $h_t = \tanh(x_t W^x + h_{t-1} W^h)$, the same job and the same $W^x = 3$. There are two things it might try, and neither works.

Try $W^h = 1$. Nothing saturates, so nothing is held: each step passes h through $\tanh$ again and shrinks it.

$$h : 0.995 \rightarrow 0.760 \rightarrow 0.641 \rightarrow 0.565 \rightarrow 0.512$$

By $T = 50$ the value is 0.17 and the gradient back to the first step is 4.6×10^{-3} . The bit is fading as it travels.

Try $W^h = 5$. Now $\tanh$ saturates and the value *is* held — h sits at 0.9999 for as long as you like. But saturation means $\tanh' \approx 0$, so look at what reaches step one:

$$\frac{\partial h_T}{\partial h_1} = \prod_{t=2}^T \tanh'(z_t) W^h = 7 \times 10^{-13} \text{ at } T = 5, \quad 10^{-149} \text{ at } T = 50.$$

	h at $T = 5$	h at $T = 50$	$\partial h_T / \partial h_1$ at $T = 50$
vanilla, $W^h = 1$	0.512	0.171	4.6×10^{-3}
vanilla, $W^h = 5$	0.9999	0.9999	10^{-149}
GRU above	0.9948	0.9927	≈ 0.998

The vanilla RNN can remember the bit or it can be trainable, but not both: holding the value requires saturation, and saturation is exactly what kills $\tanh'$. The GRU escapes the trade because its copy path multiplies by z , which can sit at 1 without any derivative being driven to zero.

Could a different activation fix it? ReLU comes close: its derivative is exactly 1 on the active side, so with $W^h = 1$ the value holds *and* the gradient passes. But that is a knife edge — at $W^h = 1.1$ the state grows every step with no saturation to bound it. We have traded vanishing for exploding, not escaped the trade. Gating works because z_t is learned and data-dependent: no fixed activation can be the identity for one unit and not another, or for fifty steps and then not.

A caveat on the construction above

That one-unit GRU leans on the 1 appearing only at the start. If 1s can occur *anywhere* and the label is still the first symbol, it breaks: on $(0, 0, 1, 0, 1)$ the gate sees $x = 1$ at $t = 3$, opens, and writes — reporting a first symbol of 1.

The unit has no way to know that step 3 is not step 1. Fixing it needs a second dimension — the same GRU with $d = 2$, so that $h_t = [h_{t,1} \ h_{t,2}]$ and the gates are 2-vectors. Give the two components different jobs:

- $h_{t,1}$ is a *flag*: its candidate is $\tanh(3)$ with no x in it, so it writes 0.995 at step 1 and then holds.
- $h_{t,2}$ is the *register*, candidate $\tanh(3x_t)$ as before.

The only new ingredient is that W_z^h is a matrix, so one component can drive the other's gate. Putting 20 in its first row and 0 in its second, with $b_z = -10$, makes both gates depend on the flag and not on x :

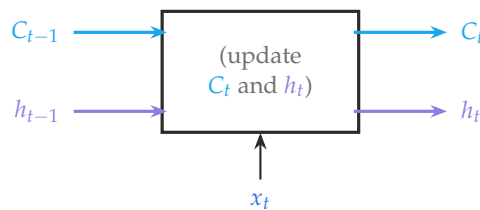
$$z_t = \sigma(20 h_{t-1,1} - 10) \implies z_t \approx 0 \text{ at } t = 1, \quad z_t \approx 1 \text{ after.}$$

So the register writes once and then copies whatever arrives later: on $(0, 0, 1, 0, 1)$ it ends at 0.0001, on $(1, 1, 0, 1, 0)$ at 0.9949, still separated (0.994 versus 0.001) at $T = 50$. The lesson is that the gate has to condition on *state*, not just input — and working out what to condition on is what training is for.

(See the notebook.)

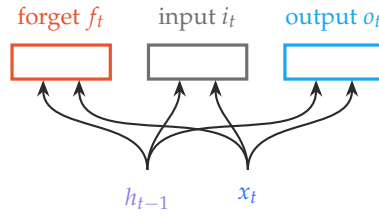
4. Long Short Term Memory (LSTM)

Like GRUs, LSTMs use gating for long-term dependencies. The difference is that an LSTM carries *two* things forward: the hidden state h_t and a separate **memory cell** C_t .



Three gates

All three are computed the same way, from h_{t-1} and x_t :



Assume batch size b and d hidden dims, with $x_t \in \mathbb{R}^{(b \times d_{\text{in}})}$ and $h_{t-1} \in \mathbb{R}^{(b \times d)}$:

$$f_t = \sigma(x_t W_f^x + h_{t-1} W_f^h + b_f)$$

$$i_t = \sigma(x_t W_i^x + h_{t-1} W_i^h + b_i)$$

$$o_t = \sigma(x_t W_o^x + h_{t-1} W_o^h + b_o)$$

Shapes, the same for all three: each gate is $\langle b \times d \rangle$, each W^x is $\langle d_{\text{in}} \times d \rangle$, each W^h is $\langle d \times d \rangle$, and each bias is $\langle 1 \times d \rangle$.

Candidate memory, and the update

Apply these gates to a candidate memory cell $\tilde{C}_t$:

$$\tilde{C}_t = \tanh(x_t W_c^x + h_{t-1} W_c^h + b_c) \quad \langle b \times d \rangle$$

Then update the memory, and read a hidden state off it:

$$C_t \leftarrow \underbrace{f_t \odot C_{t-1}}_{\text{keep}} + \underbrace{i_t \odot \tilde{C}_t}_{\text{write}}, \quad h_t \leftarrow o_t \odot \tanh(C_t)$$

Information can be stored in C_t without affecting h_t if o_t is near 0 — the cell remembers something the rest of the network is not currently reading.

5. Why gating fixes the gradient

Look at the memory update again and differentiate it:

$$C_t = f_t \odot C_{t-1} + i_t \odot \tilde{C}_t \quad \implies \quad \underbrace{\frac{\partial C_t}{\partial C_{t-1}}}_{\text{direct path}} = \text{diag}(f_t).$$

As with the GRU this is the *direct* path only — f_t , i_t and $\tilde{C}_t$ all depend on h_{t-1} , which depends on C_{t-1} , so the full derivative has further terms. But the direct path is the one that matters, and it is *diagonal*, with the forget gate as its entries. Compare the vanilla RNN, where the only path was a full Jacobian involving W^h and σ' .

Following that path over k steps, the gradient along the cell is scaled by

$$\prod_{s=t-k+1}^t f_s,$$

so if the model keeps $f \approx 1$ on some unit, that product stays near 1 and the gradient reaches back across the span essentially intact.

This is the $w = 1$ row of last lecture's table, except the model now *learns* when to sit on that row. The same idea as a residual connection in Lecture 10 — an additive path that gradient can travel without being multiplied down — running through time instead of depth.

Side by side

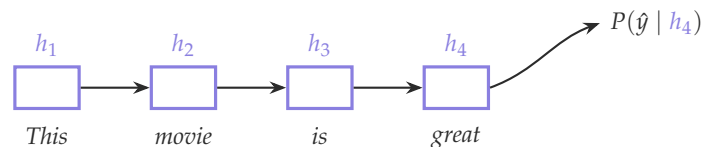
	vanilla	GRU	LSTM
carried forward	h only	h only	h and C
gates	none	reset, update	forget, input, output
how the old state survives	only via W^h and σ'	the copy path	the cell C
keep vs. write	neither: always overwrites	tied: z and $1 - z$	separate: f and i
parameters	$1 \times$	$3 \times$	$4 \times$

The parameter row counts one (W^x, W^h) pair plus a bias per gate or candidate: a vanilla cell has one such pair, a GRU three, an LSTM four.

Seen next to the vanilla cell, the gated versions are not so much adding memory as adding the *option not to overwrite*. The GRU ties its two decisions together — whatever it keeps, it does not write, and vice versa. The LSTM can do both at once, or neither. In practice the two perform similarly; the GRU is cheaper, the LSTM is the older and more standard choice.

6. (Vanilla) Attention

So far, to classify a sequence we have read the prediction off the *last* hidden state:



This puts the whole burden on h_4 : whatever the model needs to know about *This* and *movie* has to survive the trip through every later step, which is exactly the problem the last two sections were about.

Idea. Weight the hidden states, and predict from their weighted average.

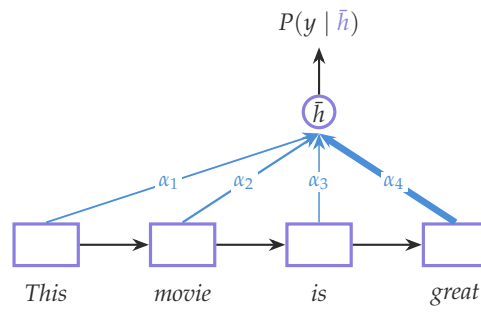
Give each position j a score with a single learned vector W_a , shared across positions:

$$a_j = \frac{W_a \cdot h_j}{\langle 1 \times d \rangle \langle d \rangle}, \quad j = 1, \dots, T.$$

Turn the scores into weights that are positive and sum to one, and average:

$$\alpha = \text{SoftMax}(a), \quad \text{i.e.} \quad \alpha_j = \frac{\exp(a_j)}{\sum_{k=1}^T \exp(a_k)}, \quad \bar{h} = \sum_{j=1}^T \alpha_j h_j.$$

The prediction is then $P(y | \bar{h})$ in place of $P(\hat{y} | h_T)$.



Every h_j now has its own direct path to the output, so neither the information nor the gradient has to pass through all the steps after j to get there. And since the α_j are computed from the hidden states, the model learns which positions to read from — here, presumably, *great*.

More next time...