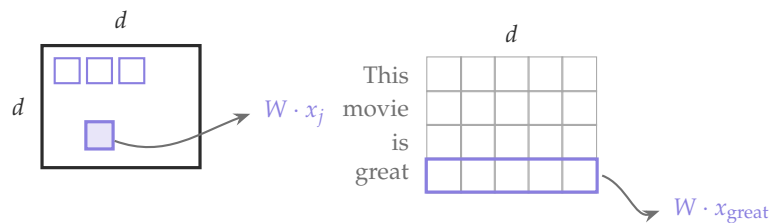


Recurrent Neural Networks

DS 4440 · LECTURE 11

1. Last few lectures: ConvNets

The same filter is applied at every position, whether the input is an image or a sentence of embeddings:



But data is often **sequentially structured**:

language genetic code
physiological data stock prices

where what comes next depends on what came before, and the thing we want to predict depends on the whole history:

$$x_t \sim P(x_t | x_{t-1} \dots x_1), \quad y_t = f(x_t \dots x_1).$$

2. The key issue

How can we model an **arbitrary length** sequence with a **finite** set of parameters?

One classical answer is the **Markov assumption**: throw away everything older than k steps,

$$P(x_t | x_{t-1} \dots x_1) = P(x_t | x_{t-1} \dots x_{t-k}),$$

which makes the input a fixed size again, at the cost of forgetting anything older. **RNNs** take a different route: they learn to induce a *fixed-length representation* of $x_1 \dots x_T$, however long the sequence is.

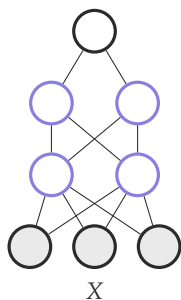
Example: learning to count

X	y
$[a \ b \ b]$	1
$[b \ b \ a \ a \ a \ b]$	0
$[a \ a \ b \ a \ b \ b \ b]$	0

$y = 1$ if $\text{count}(b)$ is even, 0 otherwise. Note the sequences have different lengths, and the answer depends on all of the input — no fixed window of the last k symbols is enough.

3. We need state

It is unclear how we would model this with an MLP or a ConvNet. MLPs do have **hidden units**:

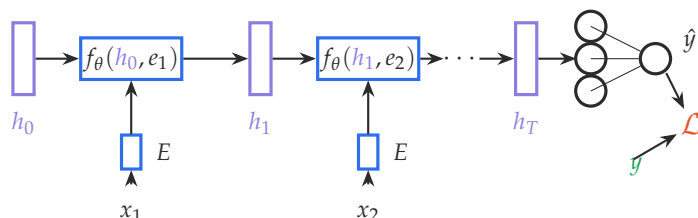


Q. Why doesn't this suffice?

A. We need to process X in *order*, updating h at each step.

4. RNNs

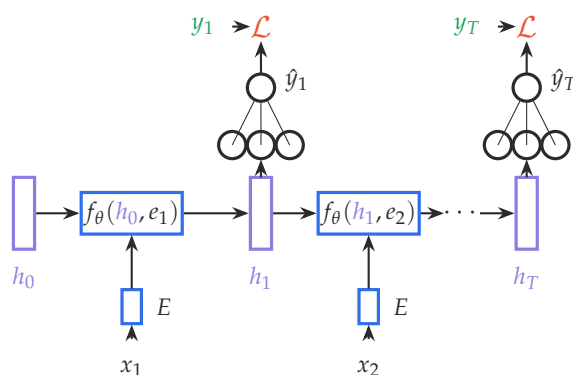
With $X = \{x_1, x_2, \dots, x_T\}$ and a single $y \in \mathbb{R}$ at the end:



f_θ is **shared** — the same function, with the same parameters, applied at every step. That is what keeps the parameter count finite however long the sequence runs.

Sequence tagging

If instead we want a $\hat{y}$ at each t , with $X = \{x_1 \dots x_T\}$ and $y = \{y_1 \dots y_T\}$, we read a prediction off every state:



Formally (batched)

$$x_t \in \mathbb{R}^{(b \times d)}, \quad h_t \in \mathbb{R}^{(b \times h)},$$

with b the batch size, d the input dimension and h the hidden size. The recurrence is one dense layer applied to the input and one to the previous state:

$$h_t \leftarrow \sigma \left(\underbrace{x_t W^x}_{\langle b \times d \rangle \langle d \times h \rangle} + \underbrace{h_{t-1} W^h}_{\langle b \times h \rangle \langle h \times h \rangle} + \underbrace{b^h}_{\langle 1 \times h \rangle} \right) \quad \langle b \times h \rangle$$

Finishing the counting example

Encode the symbols as a single number: $a \mapsto 0, b \mapsto 1$. Write p_t for the parity of the b 's seen so far. The update we need is

$$p_t = \text{XOR}(p_{t-1}, x_t) = (p_{t-1} + x_t) \bmod 2,$$

and from Lecture 3 we know one layer cannot compute XOR. So a single-unit state will not do it; we need two, with the parity recoverable from them. Write the two components of the state as $h_{t,1}$ and $h_{t,2}$, and take

$$W^x = \begin{bmatrix} 1 & 1 \end{bmatrix}, \quad W^h = \begin{bmatrix} 1 & 1 \\ -2 & -2 \end{bmatrix}, \quad b^h = \begin{bmatrix} 0 & -1 \end{bmatrix}, \quad \sigma = \text{ReLU},$$

with $h_0 = \begin{bmatrix} 0 & 0 \end{bmatrix}$. Substituting these into the recurrence, one piece at a time. The input term is

$$x_t W^x = x_t \begin{bmatrix} 1 & 1 \end{bmatrix} = \begin{bmatrix} x_t & x_t \end{bmatrix},$$

so the current symbol adds the same amount to both pre-activations. The state term is

$$h_{t-1} W^h = \begin{bmatrix} h_{t-1,1} & h_{t-1,2} \end{bmatrix} \begin{bmatrix} 1 & 1 \\ -2 & -2 \end{bmatrix} = \begin{bmatrix} h_{t-1,1} - 2h_{t-1,2} & h_{t-1,1} - 2h_{t-1,2} \end{bmatrix},$$

which is also the same in both slots, because the two columns of W^h are identical. And that repeated number is exactly the parity we are tracking:

$$p_{t-1} = h_{t-1,1} - 2h_{t-1,2}.$$

Adding the bias $\begin{bmatrix} 0 & -1 \end{bmatrix}$ and writing $s = p_{t-1} + x_t$, the pre-activation is

$$x_t W^x + h_{t-1} W^h + b^h = \begin{bmatrix} s & s-1 \end{bmatrix}, \quad \text{so} \quad h_t = \begin{bmatrix} \text{ReLU}(s) & \text{ReLU}(s-1) \end{bmatrix}.$$

Now check that this computes XOR. Both p_{t-1} and x_t are 0 or 1, so $s \in \{0, 1, 2\}$, and there are only three cases:

s	h_t	$p_t = h_{t,1} - 2h_{t,2}$	from	XOR
0	$\begin{bmatrix} 0 & 0 \end{bmatrix}$	$0 - 0 = 0$	$p = 0, x = 0$	0
1	$\begin{bmatrix} 1 & 0 \end{bmatrix}$	$1 - 0 = 1$	$p = 0, x = 1$ or $p = 1, x = 0$	1
2	$\begin{bmatrix} 2 & 1 \end{bmatrix}$	$2 - 2 = 0$	$p = 1, x = 1$	0

The second unit is what makes this work: it stays at zero until s reaches 2, and then subtracts off the overflow. That is the whole trick — one unit counts, the other cancels when the count wraps. Trace it on $\begin{bmatrix} a & b & b \end{bmatrix}$:

t	symbol	x_t	$s = p_{t-1} + x_t$	h_t	$p_t = h_{t,1} - 2h_{t,2}$	count(b)	$y_t = 1 - p_t$
0	—	—	—	$\begin{bmatrix} 0 & 0 \end{bmatrix}$	0	0	1
1	a	0	0	$\begin{bmatrix} 0 & 0 \end{bmatrix}$	0	0	1
2	b	1	1	$\begin{bmatrix} 1 & 0 \end{bmatrix}$	1	1	0
3	b	1	2	$\begin{bmatrix} 2 & 1 \end{bmatrix}$	0	2	1

p tracks parity exactly, and the label is its negation — $p = 1$ means the count is odd, $y = 1$ means it is even — so the readout just flips the bit:

$$\hat{y} = h_T W^o + b^o, \quad W^o = \begin{bmatrix} -1 \\ 2 \end{bmatrix}, \quad b^o = 1 \quad \implies \quad \hat{y} = 1 - p_T.$$

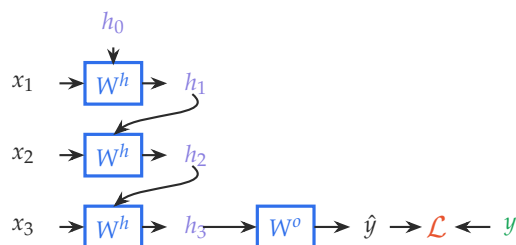
On the three rows of the table that gives 1, 0, 0 — and it keeps working for sequences of any length, with the same four numbers in W^h .

Look at what the cell is: the XOR network from Lecture 3, with one of its two inputs rewired to its own previous output. The finite parameter set handles unbounded length because the *same* cell runs again at every step.

(See the Colab notebook.)

5. Backprop through time

Take a three-step sequence and the simplest possible head, $\hat{y} = W^o h_3$:



The output weights are easy — they are used once:

$$\nabla_{W^o} \mathcal{L} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot \nabla_{W^o} \hat{y}.$$

What about $\nabla_{W^h} \mathcal{L}$?

W^h is used at *every* step, so by the fan-out rule it collects one contribution per step, and they add:

$$\nabla_{W^h} \mathcal{L} = \sum_{t=1}^T \nabla_{W^h} h_t \cdot \nabla_{h_t} \mathcal{L}$$

This is the same shared-weight situation as the convolution kernel in Lecture 9, except that the uses are strung out in *time* rather than across space.

The last term is easy:

$$\nabla_{h_3} \mathcal{L} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot \underbrace{\nabla_{h_3} \hat{y}}_{= W^o}.$$

But $\nabla_{h_2} \mathcal{L}$ is more complicated: h_2 only reaches the loss *through* h_3 , so

$$\nabla_{h_2} \mathcal{L} = (\nabla_{h_2} h_3) \cdot (\nabla_{h_3} \mathcal{L}),$$

and that first factor is a **Jacobian**:

$$\nabla_{h_2} h_3 = \begin{bmatrix} \frac{\partial h_{3,1}}{\partial h_{2,1}} & \cdots & \frac{\partial h_{3,1}}{\partial h_{2,d}} \\ \vdots & \ddots & \vdots \\ \frac{\partial h_{3,d}}{\partial h_{2,1}} & \cdots & \frac{\partial h_{3,d}}{\partial h_{2,d}} \end{bmatrix} \quad \langle d \times d \rangle$$

6. The problem: repeated multiplication

Going back one more step compounds it:

$$\nabla_{h_1} \mathcal{L} = (\nabla_{h_1} h_2)(\nabla_{h_2} h_3)(\nabla_{h_3} \mathcal{L}),$$

and this is a toy example. For a sequence of length T the gradient reaching the first step passes through $T - 1$ of these Jacobians.

From Lecture 6 we know what each one is: differentiating $h_t = \sigma(x_t W^x + h_{t-1} W^h + b^h)$ with respect to h_{t-1} gives the same route-then-gate pair as any other layer — send it back through W^h , then gate by σ' . So the product is, roughly, W^h multiplied by itself $T - 1$ times.

Repeated multiplication by the same matrix: gradients tend to **explode** or **vanish**.

Which one depends on the size of W^h . If its largest singular value exceeds 1 the product grows geometrically; if it is below 1 the product decays geometrically. Only a knife-edge value leaves the gradient intact over many steps, which is the same $(1 - 2\alpha)^t$ arithmetic from Lecture 7, now running over sequence positions instead of optimizer steps.

Gradient clipping

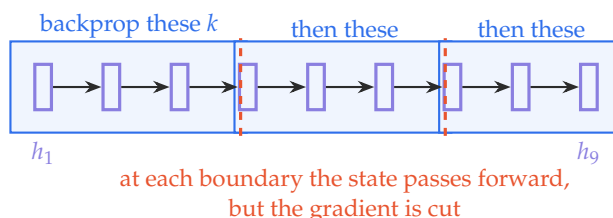
Explosions we can patch. Fix a threshold c — not to be confused with the time index t — and rescale any gradient whose norm exceeds it, leaving its direction alone:

$$\text{if } \|\nabla_\theta\| \geq c : \quad \nabla_\theta \leftarrow c \frac{\nabla_\theta}{\|\nabla_\theta\|}.$$

Note what this does *not* fix. Clipping caps a gradient that is too large; it does nothing for one that has decayed to zero.

Truncated BPTT

There is a second practical problem with the sum over t : for a long sequence we would have to keep every intermediate state alive until the backward pass reaches it, and backprop through thousands of steps to update one weight. In practice we do not.



Truncated BPTT: unroll k steps (say $k = 20$), compute the loss, backprop within that window, take a step — then carry the final state forward as the starting state of the next window, but *detached*, so no gradient flows back across the boundary.

That is exactly the `detach()` from Lecture 6: the value is kept, the history is thrown away. The forward pass still sees the whole sequence, so information can propagate arbitrarily far; only the *gradient* is cut off at k steps.

It is a bargain rather than a fix. Memory and compute per update stop growing with sequence length, which is what makes training practical. In exchange, the model cannot learn a dependency that spans more than k steps, because no gradient ever connects the two ends of it.

The same thing in one dimension

Strip the matrices away. Let the state be a single number with no input and no nonlinearity, $h_t = w h_{t-1}$. Then $\nabla_{h_1} \mathcal{L}$ picks up one factor of w per step, and the gradient reaching the first step is scaled by w^{T-1} :

	$T = 10$	$T = 50$	$T = 100$
$w = 0.9$	0.39	5.7×10^{-3}	3.0×10^{-5}
$w = 1.0$	1	1	1
$w = 1.1$	2.4	1.1×10^2	1.3×10^4

A 10% error in either direction, compounded over a hundred steps, is the difference between a gradient of 10^{-5} and one of 10^4 . Nothing is wrong with the arithmetic — this is just what repeated multiplication does.

Only $w = 1$ leaves the gradient intact over long spans. Clipping rescues the exploding row; nothing in the update rule rescues the vanishing one.

So to learn long-range dependencies we want a path through time along which the factor *is* one — something the state can carry forward unchanged, with the model deciding when to write to it and when to leave it alone. That is what gating gives us, and it is where we pick up next time.