

Recurrent Neural Networks (RNNs)

Aayush Bajaj

July 10, 2026

Contents

1	the recurrence	1
2	unrolling	2
3	backpropagation through time	2
3.1	the gradient chain	2
3.2	why the product explodes or vanishes	3
3.3	truncated bptt	4
3.4	gradient clipping	4
4	architectural variants	4
5	code: a char-level rnn from scratch	5
6	see also	6

feedforward networks eat fixed-size vectors. sequences — text, audio, sensor streams — have no fixed size, and worse, their *order* carries the meaning. the recurrent neural network solves both problems with one idea: maintain a hidden state that is updated by the same function at every time step. parameter count stops depending on sequence length, and the state becomes a lossy summary of everything seen so far (Goodfellow, Ian, 2016).

*one function, applied forever:
an rnn is a learned dynamical
system, and every pathology of
dynamical systems — fixed
points, chaos, sensitive
dependence — shows up in
training*

1 the recurrence

the vanilla (elman) rnn maintains a hidden state $h_t \in \mathbb{R}^H$ driven by inputs $x_t \in \mathbb{R}^V$:

$$h_t = \tanh(W_h h_{t-1} + W_x x_t + b), \quad o_t = W_y h_t + b_y, \quad (1)$$

with h_0 fixed (usually zero) and o_t fed to whatever head the task demands — a softmax over the vocabulary for language modelling, a single regression output at the final step for classification.

- the *same* W_h, W_x, b act at every step: parameter sharing across time, exactly analogous to a convolution's sharing across space. an rnn is a deep network whose depth is the sequence length, but whose layers are all the same layer.
- $\tanh$ keeps the state bounded in $(-1, 1)^H$; without a squashing nonlinearity the linear dynamics $h_t = W_h h_{t-1} + \dots$ would diverge or die geometrically according

to the eigenvalues of W_h . (it delays this fate; as we will see, it does not escape it.)

- the state is a bottleneck by design: h_t must compress the entire prefix $x_1, \dots, x_t$ into H numbers.

2 unrolling

recursion is just a loop, and a loop can be drawn flat. unrolling the recurrence over T steps produces an ordinary feedforward computation graph with tied weights — which is what makes the network trainable by standard backpropagation.

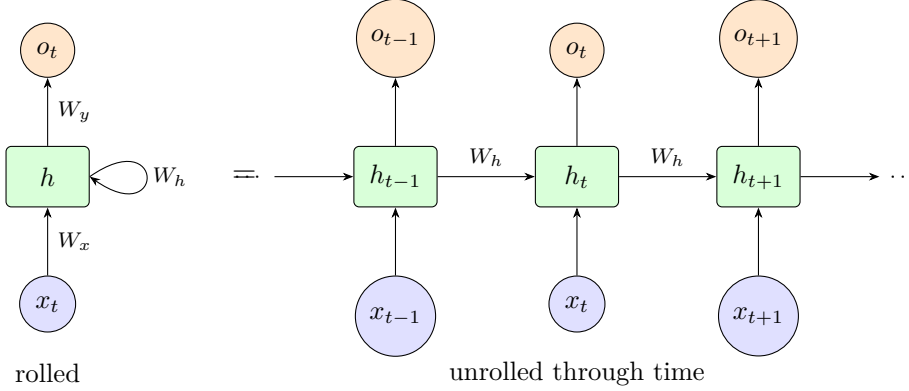


Figure 1: left: the rolled recurrence, one cell with a self-loop. right: the same network unrolled over three steps — every green box applies the identical weights.

3 backpropagation through time

bptt is nothing more than backpropagation applied to the unrolled graph, with one bookkeeping subtlety: because the weights are shared, the gradient with respect to W_h is the *sum* of the gradients from every step at which it was used.

3.1 the gradient chain

let the total loss be $L = \sum_{t=1}^T L_t$ with L_t depending on h_t through the output head. the chain rule through the recurrence gives, for the influence of an early state h_k on a late loss L_t ($k < t$):

$$\frac{\partial L_t}{\partial h_k} = \frac{\partial L_t}{\partial h_t} \prod_{j=k+1}^t \frac{\partial h_j}{\partial h_{j-1}}, \quad \frac{\partial h_j}{\partial h_{j-1}} = \text{diag}(1 - h_j^{\odot 2}) W_h, \quad (2)$$

where $1 - h_j^{\odot 2}$ is the elementwise tanh derivative (via the identity $\tanh' = 1 - \tanh^2$). in the backward pass this product is applied transposed, one factor per step:

$$\delta_{j-1} = W_h^\top \text{diag}(1 - h_j^{\odot 2}) \delta_j, \quad (3)$$

and the parameter gradients accumulate along the way:

$$\frac{\partial L}{\partial W_h} = \sum_{t=1}^T \text{diag}(1 - h_t^{\odot 2}) \delta_t h_{t-1}^\top, \quad \frac{\partial L}{\partial W_x} = \sum_{t=1}^T \text{diag}(1 - h_t^{\odot 2}) \delta_t x_t^\top,$$

where $\delta_t = \partial L / \partial h_t$ collects both the local contribution (from L_t) and the future one (from δ_{t+1} flowing back through $W_h^\top$). cost: one forward and one backward sweep, $O(T)$ time, $O(T)$ memory to store the states.

3.2 why the product explodes or vanishes

the long-range factor is a product of $t - k$ matrices of the form $D_j W_h$ with $D_j = \text{diag}(1 - h_j^{\odot 2})$, $\|D_j\| \leq 1$. take norms:

$$\left\| \prod_{j=k+1}^t D_j W_h \right\| \leq \prod_{j=k+1}^t \|D_j\| \|W_h\| \leq \sigma_{\max}(W_h)^{t-k}. \quad (4)$$

- if $\sigma_{\max}(W_h) < 1$ the long-range gradient is *guaranteed* to vanish geometrically — the right-hand side already dies, and it is only an upper bound.
- growth is governed by the spectral radius $\rho(W_h)$: over many steps the product behaves like $\rho(W_h)^{t-k}$ along the dominant eigendirection (up to the damping from the D_j). $\rho > 1$ is the signature of *exploding* gradients; $\rho \leq 1$ plus any tanh saturation (each D_j shrinks things further, since $1 - h^2 < 1$ whenever $h \neq 0$) gives *vanishing* ones.
- the uncomfortable middle: there is no setting of W_h that preserves gradient norm over hundreds of steps while also *doing* anything interesting with the state. long-range credit assignment in a vanilla rnn is structurally broken (Goodfellow, Ian, 2016).

pascanu, mikolov and bengio (2013) make this precise: sigma-max below one is sufficient for vanishing, spectral radius above one is necessary for exploding

numerically, with jacobians evaluated in the mild-activation regime (where $\tanh' \approx 1$, the regime this linearised argument describes):

```

1 import numpy as np
2
3 rng = np.random.default_rng(1)
4 H = 64
5
6 def gradient_norms(rho, steps=60):
7     W = rng.normal(0, 1, (H, H))
8     W *= rho / max(abs(np.linalg.eigvals(W)))    # rescale to spectral radius
9     # rho
10    delta = rng.normal(0, 1, H); delta /= np.linalg.norm(delta)
11    norms = []
12    for t in range(steps):
13        a = rng.normal(0, 0.3, H)                # mild pre-activations:
14        # tanh' near 1
15        delta = W.T @ ((1.0 - np.tanh(a) ** 2) * delta) # one bptt step
16        if (t + 1) % 20 == 0:
17            norms.append(np.linalg.norm(delta))
18    return norms
19
20 for rho in (0.7, 1.0, 1.3):
21     n20, n40, n60 = gradient_norms(rho)
22     print(f"rho = {rho:.1f}: |grad| after 20 steps {n20:.9.2e}, 40 steps
23           {n40:.9.2e}, 60 steps {n60:.9.2e}")

```

rho = 0.7:	grad after 20 steps	8.05e-05,	40 steps	1.40e-08,	60 steps	2.28e-12
rho = 1.0:	grad after 20 steps	6.18e-02,	40 steps	7.61e-03,	60 steps	1.41e-03
rho = 1.3:	grad after 20 steps	2.43e+01,	40 steps	7.53e+02,	60 steps	3.03e+04

note the middle row: even at $\rho = 1$ the gradient still decays, because every $\tanh'$ factor is strictly below one — the nonlinearity itself taxes the gradient at each step.

3.3 truncated bptt

full bptt on a million-token stream is both infeasible (memory grows with T) and pointless (the gradients from 10,000 steps back are zero anyway). truncated bptt processes the stream in windows of τ steps:

- **forward** τ steps, carrying h in from the previous window (so the *state* has unlimited memory),
- **backward** only within the window (so the *gradient* has horizon τ),
- **carry** the final state into the next window and repeat.

the model can therefore exploit dependencies longer than τ at inference time, but it can only *learn* dependencies shorter than τ — a bias to keep in mind when choosing the window.

3.4 gradient clipping

exploding gradients are a cliff, not a slope: a single step through a high-curvature wall can catapult the weights into garbage. the standard fix is embarrassingly blunt — if the gradient is too big, shrink it:

$$g \leftarrow g \cdot \min\left(1, \frac{c}{\|g\|}\right), \quad (5)$$

which preserves direction while capping magnitude (elementwise clipping to $[-c, c]$, as in the code below, is the cruder cousin that also works fine in practice). clipping handles *exploding* gradients only; vanishing ones need an architectural fix — that is the *lstm*’s job.

4 architectural variants

- **bidirectional rnns** run two chains, one left-to-right and one right-to-left, and concatenate the states: $h_t = (\vec{h}_t, \overleftarrow{h}_t)$. the representation of token t then conditions on the *whole* sequence — essential for tagging tasks where the right context disambiguates, unusable for autoregressive generation (you cannot condition on a future you are about to produce).
- **sequence-to-sequence (encoder-decoder)**: one rnn consumes the source sequence into a final state (the “thought vector”); a second rnn, initialised from that state, emits the target sequence token by token, feeding each output back in as the next input. this architecture carried machine translation until attention arrived (Sutskever, Ilya and Vinyals, Oriol and Le, Quoc V., 2014) — and its fixed-size bottleneck is precisely what attention (and then *transformers*) removed.

- **deep rnns** stack cells vertically: the state sequence of layer ℓ is the input sequence of layer $\ell + 1$. depth in space and depth in time are independent axes.

5 code: a char-level rnn from scratch

everything above, in about sixty lines: one-hot characters in, softmax over next character out, bptt over a sliding window with the state carried across windows, adagrad updates, elementwise clipping.

a miniature of karpathy's min-char-rnn, rewritten and rerun for this page

```

1 import numpy as np
2
3 rng = np.random.default_rng(0)
4
5 corpus = "the quick brown fox jumps over the lazy dog. " * 4
6 chars = sorted(set(corpus))
7 V = len(chars)
8 c2i = {c: i for i, c in enumerate(chars)}
9
10 H, T, lr = 48, 24, 0.1                # hidden size, bptt window,
    ↪ adagrad rate
11 Wx = rng.normal(0, 0.01, (H, V))      # input -> hidden
12 Wh = rng.normal(0, 0.01, (H, H))      # hidden -> hidden
13 Wy = rng.normal(0, 0.01, (V, H))      # hidden -> output
14 bh, by = np.zeros(H), np.zeros(V)
15
16 def loss_and_grads(inputs, targets, h):
17     xs, hs, ps = {}, {-1: h}, {}
18     loss = 0.0
19     # --- forward: unroll T steps ---
20     for t, (ci, ct) in enumerate(zip(inputs, targets)):
21         xs[t] = np.zeros(V); xs[t][ci] = 1.0
22         hs[t] = np.tanh(Wx @ xs[t] + Wh @ hs[t - 1] + bh)
23         z = Wy @ hs[t]
24         ps[t] = np.exp(z - z.max()); ps[t] /= ps[t].sum()
25         loss += np.log(ps[t][ct])
26     # --- backward: bptt ---
27     g = {k: np.zeros_like(v) for k, v in
28          dict(Wx=Wx, Wh=Wh, Wy=Wy, bh=bh, by=by).items()}
29     dh_next = np.zeros(H)
30     for t in reversed(range(len(inputs))):
31         dz = ps[t].copy(); dz[targets[t]] -= 1.0    # softmax + nll
32         ↪ gradient
33         g["Wy"] += np.outer(dz, hs[t]); g["by"] += dz
34         dh = Wy.T @ dz + dh_next                  # from output AND from
35         ↪ the future
36         da = (1.0 - hs[t] ** 2) * dh               # through tanh: diag(1
37         ↪ - h^2)
38         g["Wx"] += np.outer(da, xs[t]); g["bh"] += da
39         g["Wh"] += np.outer(da, hs[t - 1])
40         dh_next = Wh.T @ da                        # W_h^T carries credit
41         ↪ backwards
42     for k in g:
43         np.clip(g[k], -5, 5, out=g[k])             # gradient clipping
44     return loss, g, hs[len(inputs) - 1]
45
46 # --- training loop: adagrad over sliding windows (truncated bptt) ---
47 params = dict(Wx=Wx, Wh=Wh, Wy=Wy, bh=bh, by=by)
48 mem = {k: np.zeros_like(v) for k, v in params.items()}
49 data = [c2i[c] for c in corpus]
50 h, pos = np.zeros(H), 0
51 for step in range(3001):
52     if pos + T + 1 >= len(data):
53         h, pos = np.zeros(H), 0                    # wrap: reset carry
54     loss, g, h = loss_and_grads(data[pos:pos+T], data[pos+1:pos+T+1], h)

```

```

51     if step % 500 == 0:
52         print(f"step {step:>4}  loss/char {loss/T:.3f}  (uniform =
           ↳ {np.log(V):.3f})")
53     for k in params:
54         mem[k] += g[k] ** 2
55         params[k] -= lr * g[k] / np.sqrt(mem[k] + 1e-8) # adagrad
56     pos += T
57
58 # --- sample from the trained model ---
59 h, ci, out = np.zeros(H), c2i["t"], "t"
60 for _ in range(60):
61     x = np.zeros(V); x[ci] = 1.0
62     h = np.tanh(Wx @ x + Wh @ h + bh)
63     z = Wy @ h
64     p = np.exp(z - z.max()); p /= p.sum()
65     ci = int(p.argmax()) # greedy decode
66     out += chars[ci]
67 print("sample:", out)

```

```

step    0  loss/char 3.332  (uniform = 3.332)
step  500  loss/char 0.015  (uniform = 3.332)
step 1000  loss/char 0.005  (uniform = 3.332)
step 1500  loss/char 0.003  (uniform = 3.332)
step 2000  loss/char 0.003  (uniform = 3.332)
step 2500  loss/char 0.002  (uniform = 3.332)
step 3000  loss/char 0.002  (uniform = 3.332)
sample: the quick brown fox jumps over the lazy dog. the quick brown

```

the loss starts at exactly $\log V$ (uniform guessing over 28 characters) and collapses to near zero: with a 24-character bptt window and a 45-character period, the corpus is comfortably memorisable, and greedy decoding regenerates it verbatim — including crossing the sentence boundary, which requires state carried further than any single window. a worked application at scale lives on the [sentiment analysis](#) child page.

6 see also

- [long short-term memory](#) — the architectural cure for the vanishing gradient
- [sentiment analysis](#) — an rnn earning its keep on real text
- [transformers](#) — what replaced recurrence with attention
- [feedforward deep neural networks](#) — the acyclic special case
- [perceptron](#) — the single unit all of this is built from

references

Goodfellow, Ian (2016). *Deep Learning*, MIT Press.

Sutskever, Ilya and Vinyals, Oriol and Le, Quoc V. (2014). *Sequence to Sequence Learning with Neural Networks*.