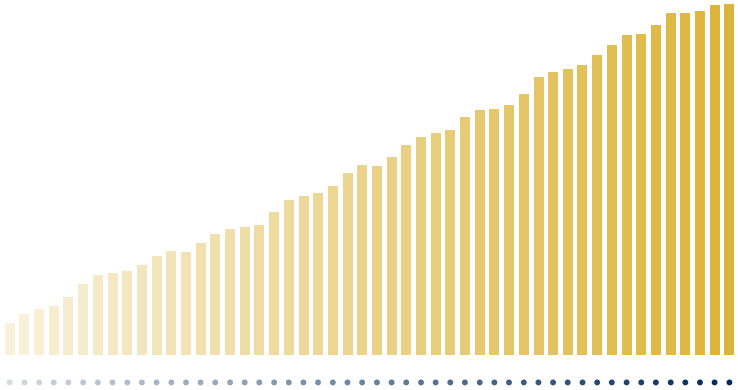


UNIT 03 · CHAPTER 4

Backpropagation Through Time

CSE 30124 · Introduction to Artificial Intelligence
William Theisen · University of Notre Dame

Training an RNN

Last chapter's **recurrent neural network** read the Victory March one word at a time, mixing each word with a hidden state carried over from the word before:

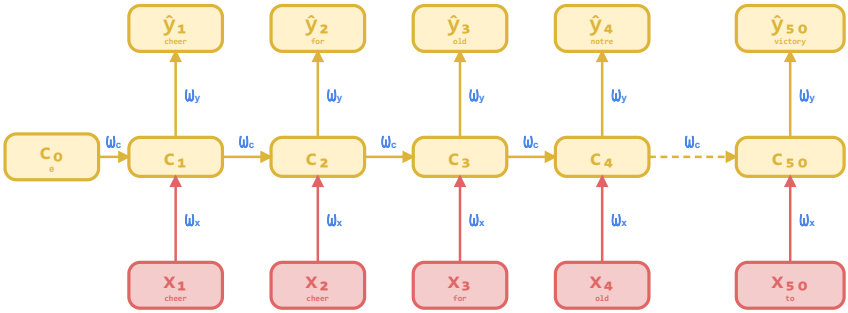
[4.1] the new hidden state

$$c_t = \tanh(x_t W_x^T + c_{t-1} W_c^T + b_c)$$

[4.2] the prediction

$$\hat{y}_t = \mathrm{softmax}(c_t W_y^T + b_y)$$

Unfortunately its weights were made up, so it forgot how the song started within a few words. To learn real ones, this chapter gives the network more room: the whole chorus, **51** words and **41** different ones, tokenized the same way, and a hidden state of **16** numbers. Each word comes in one-hot, so $x_t W_x^T$ picks out the word's column of W_x, which makes W_x the embedding table, learned with everything else. At every word the job is to **predict the next one**:



Loss Over a Sequence

Pictured on the last page is the network unrolled over the chorus, one copy of the cell for each of the `50` predictions. They aren't separate networks, every copy uses the same three weight matrices. Drawn this way **an RNN is just a very deep feed-forward network**, `50` layers deep, that happens to reuse its weights in every layer. So we should be able to train it the way we train any other network, with a loss and backpropagation.

Each copy makes a prediction, so each copy gets a loss. Copy `t` gives a probability to each of the `41` words, and as with logistic regression we charge it the negative log of the probability it gave the word that actually came next, y_t. The loss for the whole chorus is the average over all `50`:

each word's loss, and their average

[4.3] $$\mathcal{L}_t = -\log \hat{y}_t[y_t] \qquad \mathcal{L} = \frac{1}{50}\sum_{t=1}^{50} \mathcal{L}_t$$

A model that knows nothing at all gives every word the same chance, `1` in `41`, and pays $-\log(1/41) = \log 41 = 3.714$ on every word. That makes a handy yardstick for a model that hasn't learned anything yet, such as ours.

Computing the Loss

Here's the loss in PyTorch, for our RNN with random starting weights (numpy draws them, so they come out the same everywhere):

```
import numpy as np
import torch
song = """cheer cheer for old notre dame wake up the
echoes cheering her name send a volley cheer on
high shake down the thunder from the sky what
though the odds be great or small old notre dame
will win over all while her loyal sons and
daughters march on to victory"""
words = song.split()
vocab = list(dict.fromkeys(words))  # first appearance
seq = [vocab.index(w) for w in words]
x, y = seq[:-1], seq[1:]       # each word, the next
V, H, T = len(vocab), 16, len(x)
rng = np.random.default_rng(0)  # same start as before
f = lambda d, n: torch.tensor(rng.normal(0, d, n)).float()
Wx = f(V ** -0.5, (H, V))
Wc = f(H ** -0.5, (H, H))
Wy = f(H ** -0.5, (V, H))
bc, by = torch.zeros(H), torch.zeros(V)
c, loss = torch.zeros(H), 0
for t in range(T):
    c = torch.tanh(Wx[:, x[t]] + c @ Wc.T + bc)
    o = c @ Wy.T + by
    p = torch.softmax(o, dim=0)
    loss -= torch.log(p[y[t]])
print(V, T, round(loss.item() / T, 3),
      round(np.log(V), 3))
```

OUTPUT

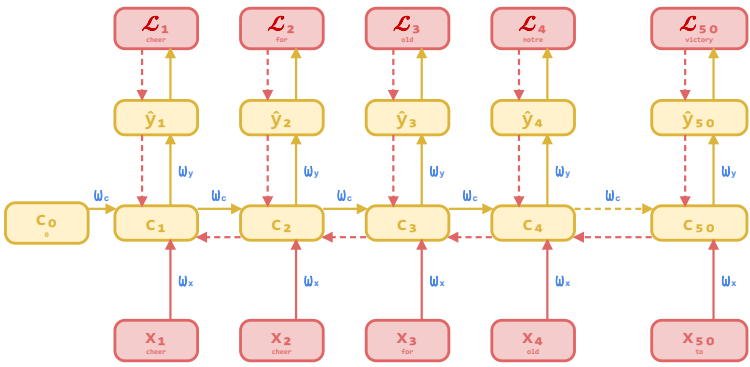
```
41 50 3.719 3.714
```

The untrained RNN scores `3.719`, a hair worse than knowing nothing, which is about what random weights should do.

Backpropagation Through Time

To train it we need the gradient of $\mathcal{L}$ for W_x, W_c and W_y. We get it by running the chain rule backward through the unrolled network, from the last word to the first, which is called **backpropagation through time** (BPTT). Really **it's just backpropagation on a very long network**.

The output end is familiar. Through a softmax, the cross-entropy's derivative is $\hat{y}_t - y_t$ (with y_t one-hot), the same "prediction minus truth" as logistic regression. What's new is that each c_t (pictured below) feeds its own prediction and the next copy, so its gradient comes from both:



from its own prediction, and from the next copy

$$[4.4] \quad \frac{\partial \mathcal{L}}{\partial c_t} = (\hat{y}_t - y_t) W_y + \frac{\partial \mathcal{L}}{\partial z_{t+1}} W_c$$

back through the tanh

$$[4.5] \quad \frac{\partial \mathcal{L}}{\partial z_t} = \frac{\partial \mathcal{L}}{\partial c_t} \odot \left(1 - (c_t)^2\right)$$

Gradients of Shared Weights

In those two equations, z_t is what goes into the tanh, $1 - (c_t)^2$ is the tanh's derivative, and we've left out the $1/50$ to keep things readable. The last copy has no next copy, so the gradient starts there with just its own loss and gets handed back one word at a time, the same way the forward pass handed the hidden state forward.

Once we have $\partial \mathcal{L}/\partial z_t$ at every word, the weights are nearly done, with one twist. w_c is used by all 50 copies, so it gets a gradient from all 50, and the chain rule says to add them up. The same goes for w_x and w_y:

[4.6]

one term per copy, added up

$$\frac{\partial \mathcal{L}}{\partial w_c} = \sum_{t=1}^{50} \left(\frac{\partial \mathcal{L}}{\partial z_t} \right)^{\mathsf{T}} c_{t-1} \qquad \frac{\partial \mathcal{L}}{\partial w_x} = \sum_{t=1}^{50} \left(\frac{\partial \mathcal{L}}{\partial z_t} \right)^{\mathsf{T}} x_t$$

$$\frac{\partial \mathcal{L}}{\partial w_y} = \sum_{t=1}^{50} (\hat{y}_t - y_t)^{\mathsf{T}} c_t$$

Each term is a one-layer gradient of the kind we've seen since logistic regression, an error times an input, and there's one per copy. Adding them is **what makes it one network rather than 50**. w_c moves in whichever direction helps all the words, on average.

Over the page is all of it in PyTorch, by hand, repeated a thousand times with $\alpha = 0.5$:

Training on the Victory March

```
import numpy as np
import torch
song = """cheer cheer for old notre dame wake up the
echoes cheering her name send a volley cheer on high
shake down the thunder from the sky what though the
odds be great or small old notre dame will win over all
while her loyal sons and daughters march on to victory"""
vocab = list(dict.fromkeys(song.split()))
s = [vocab.index(w) for w in song.split()]
V, H, T = len(vocab), 16, len(s) - 1
X, Y = torch.eye(V)[s[:-1]], torch.eye(V)[s[1:]]
rng = np.random.default_rng(0)
f = lambda d, n: torch.tensor(rng.normal(0, d, n)).float()
Wx, Wc = f(V ** -0.5, (H, V)), f(H ** -0.5, (H, H))
Wy = f(H ** -0.5, (V, H))
bc, by = torch.zeros(H), torch.zeros(V)
for step in range(1000):  # about 30 s in the browser
    c, Zx = [torch.zeros(H)], X @ Wx.T + bc   # forward
    for t in range(T):
        c.append(torch.tanh(Zx[t] + c[t] @ Wc.T))
    C = torch.stack(c)                  # c₀ to c_T
    P = torch.softmax(C[1:] @ Wy.T + by, dim=1)
    dO = P - Y                         # y^ - y, every step
    dC, dz, dnext = dO @ Wy, [], torch.zeros(H)
    for t in reversed(range(T)):        # backward
        dz.append((dC[t] + dnext) * (1 - C[t + 1] ** 2))
        dnext = dz[-1] @ Wc            # to step t-1
    dZ = torch.stack(dz[::-1])
    G = [dZ.T @ X, dZ.T @ C[:-1], dZ.sum(dim=0),
         dO.T @ C[1:], dO.sum(dim=0)]
    Wx, Wc, bc, Wy, by = [W - 0.5 * g / T  # alpha 0.5
        for W, g in zip((Wx, Wc, bc, Wy, by), G)]
loss = -torch.log((P * Y).sum(dim=1)).mean()
right = P.argmax(dim=1) == Y.argmax(dim=1)
print(round(loss.item(), 4), int(right.float().sum()),
      "of", T)
```

OUTPUT

```
0.0145 50 of 50
```

Gradient Checking

After `1,000` steps the loss is down from `3.719` to `0.015` and the RNN gets every one of the `50` next words right. That includes both "dame"s, "wake" after the first and "will" after the second, which **it can only do by remembering what came before**. (Memorizing one chorus with `1,625` weights isn't much of an achievement, but it's a good sign the gradients are right.)

Only a sign, though. A network can learn a fair amount with slightly wrong gradients, and with this many indices it's easy to get one wrong. The way to be sure is to compare against brute force, called **gradient checking**. Nudging a single entry of W_c up and down by `0.000001` in the untrained RNN and measuring how much the loss changes gives a slope of `-0.01508333`. BPTT gives `-0.01508333` too.

Writing the backward loop by hand is a good way to see what BPTT does and a bad way to get it right. As we saw in Backpropagation, PyTorch records every operation of the forward pass and can run the chain rule back through them for us, and an unrolled RNN is just a lot of operations. Over the page is the same loss with W_c marked as needing a gradient, and the same nudge as a check:

BPTT in PyTorch

```
import numpy as np
import torch
song = """cheer cheer for old notre dame wake up the
echoes cheering her name send a volley cheer on high
shake down the thunder from the sky what though the
odds be great or small old notre dame will win over all
while her loyal sons and daughters march on to victory"""
vocab = list(dict.fromkeys(song.split()))
s = [vocab.index(w) for w in song.split()]
V, H, T = len(vocab), 16, len(s) - 1
rng = np.random.default_rng(0)       # the same start
f = lambda d, n: torch.tensor(rng.normal(0, d, n))
Wx, Wc = f(V ** -0.5, (H, V)), f(H ** -0.5, (H, H))
Wy = f(H ** -0.5, (V, H))
bc, by = torch.zeros(H).double(), torch.zeros(V).double()
def loss_of(Wc):
    c, loss = torch.zeros(H).double(), 0
    for t in range(T):
        c = torch.tanh(Wx[:, s[t]] + c @ Wc.T + bc)
        p = torch.softmax(c @ Wy.T + by, dim=0)
        loss = loss - torch.log(p[s[t + 1]])
    return loss / T
Wc.requires_grad_()
loss_of(Wc).backward()                # BPTT, for us
e = torch.zeros(H, H).double()
e[2, 3] = 1e-6                         # one weight
nudged = (loss_of(Wc + e) - loss_of(Wc - e)) / 2e-6
print(round(Wc.grad[2, 3].item(), 8),
      round(nudged.item(), 8))
```

OUTPUT

```
-0.01508333 -0.01508333
```

Both agree with our hand-written BPTT to eight decimal places, so from here on **`loss.backward()` does BPTT for us**.

Truncated BPTT

Keeping every step of the forward pass around for the backward pass takes memory, and for a long text (a book, say) far too much. So in practice the text is cut into chunks of a few dozen words, the hidden state is carried from chunk to chunk as usual, and the gradient is stopped at the start of each chunk with `c.detach()`. This is called **truncated BPTT**.

We tried it on the chorus, stopping the gradient every k words and otherwise training exactly as before:

k	1	2	3	5	10	50 (all)
loss	0.018	0.022	0.016	0.018	0.015	0.015
right	50	50	50	50	50	50

Even k = 1, where the gradient never crosses from one word to the one before, gets all `50`. The hidden state still carries the song forward (even random weights give the two "dame"s different hidden states), and **our chorus never needs the network to learn to hold on to something far back**, so the gradient never has to go far.

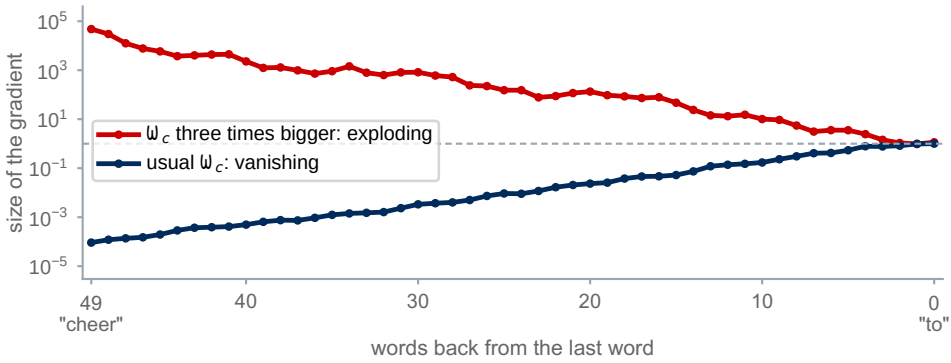
Unfortunately, most text does. Last chapter our made-up RNN had forgotten how the song started by the ninth word, and teaching a network to hold on longer means sending the gradient back at least that far.

Vanishing Gradients

Each word we walk back multiplies the gradient by the tanh's derivative, never more than `1`, and by w_c. Following the last word's loss back to word k repeats that pair of factors (D_t is a diagonal matrix of the $1 - (c_t)^2$):

```
Missing $ inserted.
l.5 ...} \cdots D_{k+1} \eqfam{wblue}{\Omega_{c}}}
```

Pictured below is its size for our untrained RNN:



With the usual starting weights (navy) the gradient is `1.00` at the last word. Ten words back it's `0.17`, twenty words back `0.023`, and at the first "cheer" it's `0.000092`, about `11,000` times smaller. This is called the **vanishing gradient**. Any deep network can have it, but an RNN is as deep as its sequence is long, so **an RNN struggles to learn anything that depends on words far back**.

Exploding Gradients

The red curve on the last page is the same network with W_c three times bigger. Now the factors tend to grow the gradient rather than shrink it: about `10` at ten words back, `134` at twenty, and about `48,000` at the first "cheer". That's an **exploding gradient**, and it's no better. Multiply `48,000` by any sensible learning rate and the weights land somewhere wild.

Which one we get depends on whether each word's factor tends to be below or above `1`, and either way it gets raised to a big power. A factor of `0.8` per word leaves `0.00002` after `49` words, and `1.2` per word gives `7,584`. Unfortunately, **no setting is safe for long sequences**, since anything a little away from exactly `1` ends up enormous or negligible. (This is extremely approximate, since the factors are matrices that change from word to word, but it's the right idea.)

The two problems also have the same cause. The gradient has to pass through W_c and a tanh at every single word, with no way around them.

Gradients Through a Gate

Last chapter's fix for forgetting was a gate, a vector z_t of numbers between 0 and 1 deciding how much of the old hidden state to keep. It turns out to be a way around for the gradient too:

[4.8] last chapter's gate

$$c_t = z_t \odot c_{t-1} + (1 - z_t) \odot \tilde{c}_t$$

straight across, plus the paths through the candidate

[4.9]

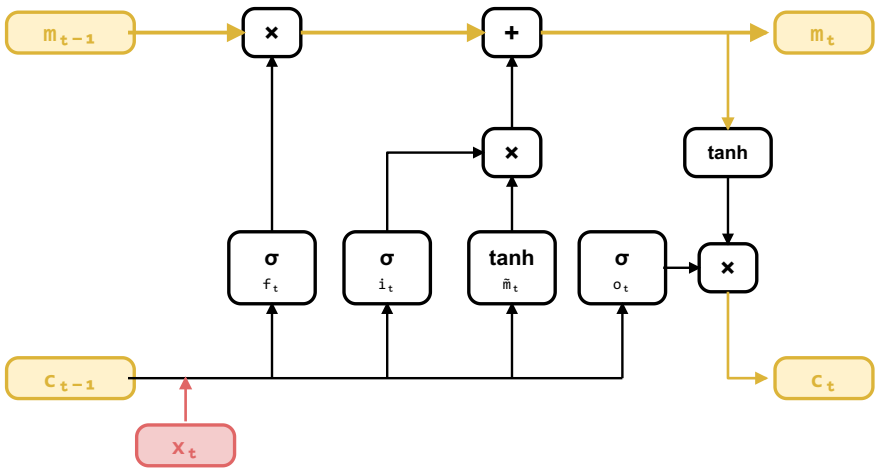
$$\frac{\partial c_t}{\partial c_{t-1}} = \mathrm{diag}(z_t) + \cdots$$

Walking back one word, c_{t-1} reaches c_t two ways, through the candidate $\tilde{c}_t$ (with its W_c and tanh, as before) and straight across, multiplied only by z_t. **The straight path has no weight matrix and no tanh on it.** A gate of 0.9 at every word leaves $0.9^{49} = 0.0057$ of the gradient after 49 words, about 60 times what reached the first "cheer" in our RNN, and a gate of 0.99 leaves 0.61.

The catch, as we saw, is that a gate stuck near 1 remembers everything just as hard. So the gate has to be learned as well, from the word and the hidden state, and that's what LSTMs and GRUs do.

Long Short-Term Memory

The Long Short-Term Memory unit (LSTM, Hochreiter and Schmidhuber, **1997**) was built around that straight path. Alongside the hidden state it keeps a second vector, the **cell state** m_t, which is only ever changed by a multiplication and an addition, along the top of the picture:



[4.10] the new cell state

$$m_t = f_t \odot m_{t-1} + i_t \odot \tilde{m}_t$$

f_t is how much of the old cell state to keep, and $i_t \odot \tilde{m}_t$ is the new information being written in. Following the gradient back along that top line, $\partial m_t / \partial m_{t-1}$ is just f_t. There's no W_c and no tanh in the way, so when f_t is close to **1**, **the gradient passes back through the cell state almost untouched**.

The LSTM's Gates

The forget gate is one of three. Each gate is a sigmoid layer looking at the current word and the previous hidden state, so each is a vector of numbers between `0` and `1` that multiplies something elementwise:

[4.10]

forget gate: how much of the old cell to keep

$$f_t = \sigma\left(x_t W_{fx}^\top + c_{t-1} W_{fc}^\top + b_f\right)$$

input gate: how much of the candidate to write

$$i_t = \sigma\left(x_t W_{ix}^\top + c_{t-1} W_{ic}^\top + b_i\right)$$

candidate: what to write

$$\tilde{m}_t = \tanh\left(x_t W_{gx}^\top + c_{t-1} W_{gc}^\top + b_g\right)$$

output gate: how much of the cell to show

$$o_t = \sigma\left(x_t W_{ox}^\top + c_{t-1} W_{oc}^\top + b_o\right)$$

the new cell state

$$m_t = f_t \odot m_{t-1} + i_t \odot \tilde{m}_t$$

the new hidden state

$$c_t = o_t \odot \tanh(m_t)$$

The hidden state is now a filtered view of the cell state, squashed by a tanh and scaled by the output gate. Predictions only ever see c_t, so **the cell state is free to hold on to things the current prediction doesn't need**.

That's four layers where the RNN had one, so four times the recurrent weights, `3,712` for our `16` units against the RNN's `928`.

An LSTM on the Victory March

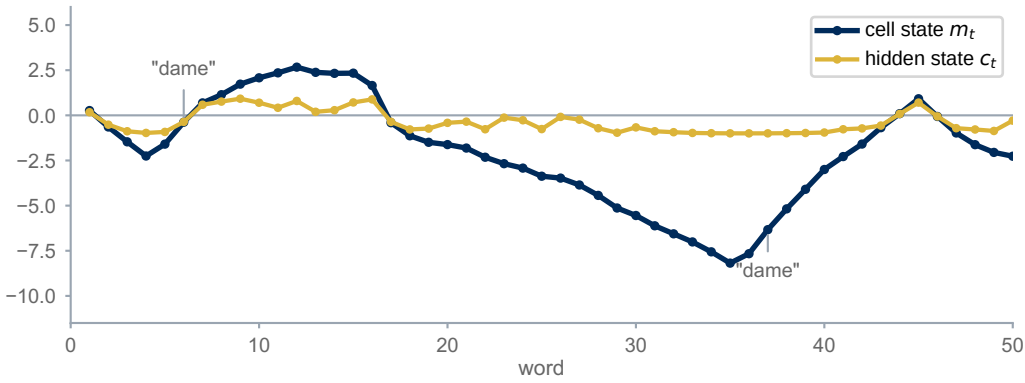
We trained an LSTM with `16` units on the chorus the same way as the RNN (`1,000` steps, $\alpha = 0.5$), and it gets all `50` next words right as well. Here's one of its units the two times the chorus sings "old notre dame":

word		f	i	$\tilde{m}$	o	m	c
4	old	0.91	0.94	−0.97	1.00	−2.25	−0.97
5	notre	0.93	0.94	0.53	1.00	−1.59	−0.92
6	dame	0.79	0.95	0.92	1.00	−0.38	−0.36
35	old	0.97	0.89	−0.96	1.00	−8.18	−1.00
36	notre	0.97	0.89	0.31	1.00	−7.66	−1.00
37	dame	0.91	0.88	0.77	1.00	−6.33	−1.00

The two "dame"s get exactly the same input, so the only way to tell them apart is what the cell remembers. In this unit **the cell state is `-0.38` at the first "dame" and `-6.33` at the second**, and the hidden state passes that on as `-0.36` and `-1.00`. Notice the forget gate, `0.97` on the second "old" and "notre", keeping almost all of the old cell state each word.

Memory in the Cell State

Pictured below is the same unit across the whole chorus. The cell state wanders well outside `-1` to `1`, while the hidden state, squashed by its tanh, never does:

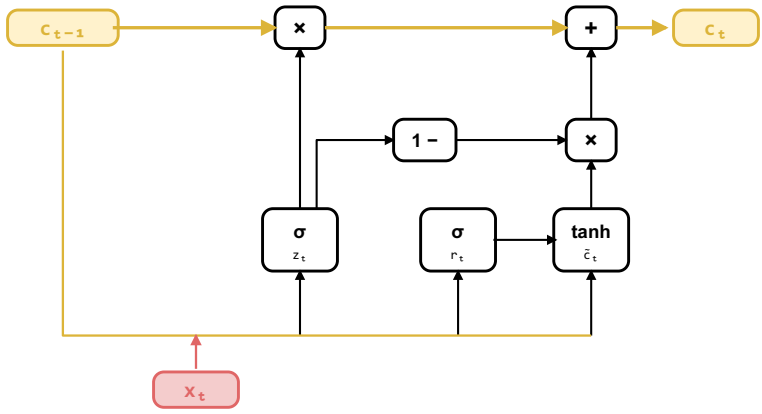


From the third "cheer" (word `17`) to the second "old" (word `35`) the cell state sinks by a little under half a step every word, so this unit seems to be keeping track of how far into the song we are. Its forget gates from word `27` to word `36` multiply to `0.66`, so a gradient sent back along its cell state over those `10` words keeps two thirds of its size. The untrained RNN's shrank by a factor of about `6` over the same distance.

(On a chorus this short the plain RNN gets every word right too, so this shows how the LSTM remembers, not that it needs to.)

Gated Recurrent Units

All those weights make the LSTM slow. The **Gated Recurrent Unit** (GRU, Cho and others, 2014) is last chapter's gate made learnable, with no separate cell state. An update gate z_t does the job of the forget and input gates, and a reset gate r_t decides how much of the old state the candidate gets to see:



$$
\begin{aligned}
z_t &= \sigma\left(x_t W_{zx}^\top + c_{t-1} W_{zc}^\top + b_z\right) \\
r_t &= \sigma\left(x_t W_{rx}^\top + c_{t-1} W_{rc}^\top + b_r\right) \\
\tilde{c}_t &= \tanh\left(x_t W_x^\top + (r_t \odot c_{t-1}) W_c^\top + b_c\right) \\
c_t &= z_t \odot c_{t-1} + (1 - z_t) \odot \tilde{c}_t \qquad [4.11]
\end{aligned}
$$

When z_t is near 1, c_t is just c_{t-1} and the gradient passes straight across. That's 2,784 recurrent weights for our 16 units, three times the RNN's, and on many tasks **GRUs do about as well as LSTMs**.

Gradient Clipping

Unfortunately, **gates only fix half the problem**. The cell state's path keeps the gradient from vanishing, but every other path still goes through weight matrices, and LSTMs are still prone to exploding gradients. While ours trained, the gradient's size was usually around `0.08`, and at step `199` it jumped to `3.09`, nearly forty times that.

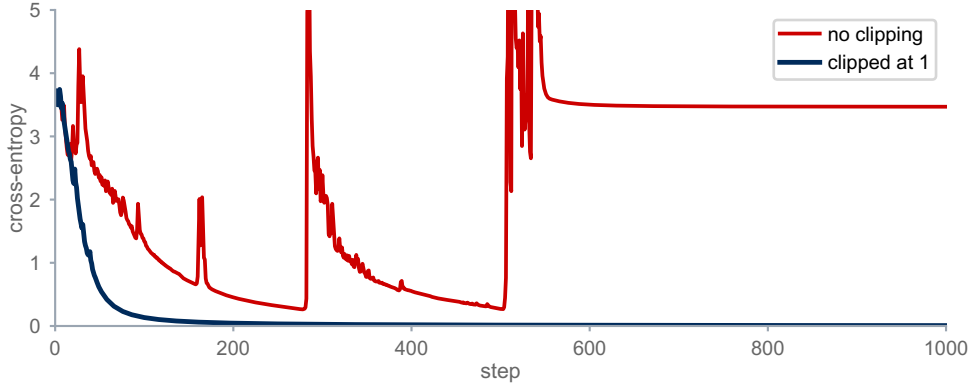
The usual fix is blunt. If the gradient's size (its norm, over all the weights at once) is bigger than some ceiling τ, we scale it down to τ before taking the step:

if the gradient's norm is over the ceiling τ, scale it down to τ

[4.12] $$g \leftarrow \tau \frac{g}{\|g\|} \qquad \text{if } \|g\| > \tau$$

This is called **gradient clipping**. It keeps the gradient's direction and only limits how far one step can go. Our plain RNN shows the difference best. With $\alpha = 1$, twice the learning rate that worked, its loss is down to `0.26` by step `278`, then lurches back up to `6`. It recovers, and around step `507` the loss shoots from `0.3` to over `4`, with gradients of `17`, `45`, `81` and `194` in the next `40` steps. After `1,000` steps it gets `6` of the `50` words right.

Gradient Clipping in Practice



Clipped at `1`, the same run gets all `50`, and only `7` of its `1,000` steps ever needed clipping. **Clipping does nothing for vanishing gradients** though, since a tiny gradient is already under the ceiling.

Gradient Clipping in PyTorch

In practice clipping is a single line, between computing the gradients and taking the step. Here is a PyTorch RNN learning to add up 50 numbers, clipped at 1:

```
import torch
from torch import nn
torch.manual_seed(0)
rnn = nn.RNN(input_size=1, hidden_size=16,
             batch_first=True)
head = nn.Linear(16, 1)
params = list(rnn.parameters()) + list(head.parameters())
optimizer = torch.optim.SGD(params, lr=1.0)
x = torch.randn(8, 50, 1)  # 8 sequences of 50 steps
y = x.sum(dim=1)           # predict each one's sum
for step in range(3):
    out, h = rnn(x)
    loss = ((head(out[:, -1]) - y) ** 2).mean()
    optimizer.zero_grad()
    loss.backward()                  # BPTT happens here
    norm = nn.utils.clip_grad_norm_(params, max_norm=1.0)
    optimizer.step()
    print(step, round(loss.item(), 2),
          round(norm.item(), 2))
```

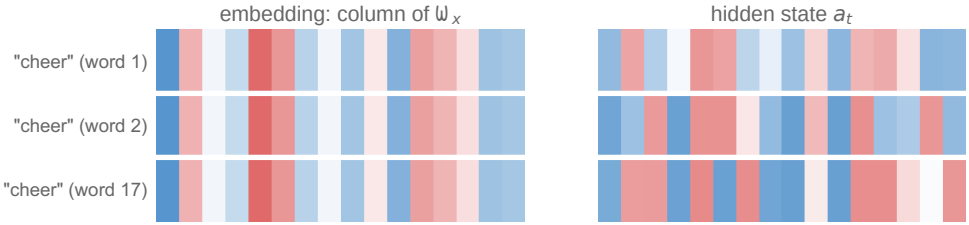
OUTPUT

```
0 29.75 7.91
1 23.29 8.83
2 14.47 9.76
```

`clip_grad_norm_` returns the norm before clipping, about 8 to 10 here (desktop PyTorch draws different random numbers, so its numbers differ), so every step is cut to a length of 1. `loss.backward()` runs BPTT for us, which after the last few pages is probably for the best.

Static Embeddings

Whatever the cell, the input for a word is the same. In our RNN it's a column of W_x, the word's embedding. "cheer" appears three times in the chorus, and each time it gets exactly the same `16` numbers, whether it's the first word of the song or the one in "send a volley cheer on high". An embedding that doesn't depend on the words around it is called a **static embedding**.

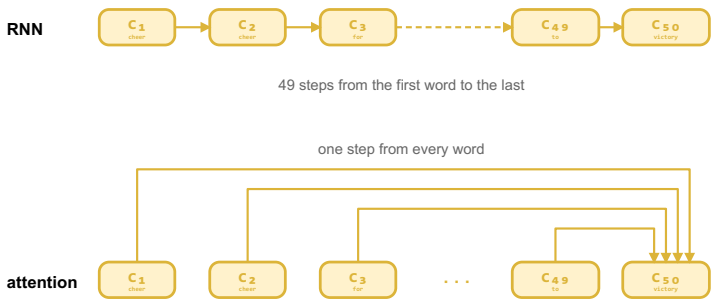


The only context the RNN has is its hidden state. After each of the three "cheer"s the trained RNN's hidden states are quite different from each other (cosine similarities of only `0.34` to `0.40`), and they need to be, since the next words are "cheer", "for" and "on". So **all the context lives in c_t**, `16` numbers that have to carry everything the network knows about the song so far.

Attention

That's a lot to ask of 16 numbers. To predict "victory", anything the RNN needs from "cheer, cheer for old notre dame" has to survive 49 updates of the hidden state, and the gradient that would teach it to keep it has to make the same 49 steps back.

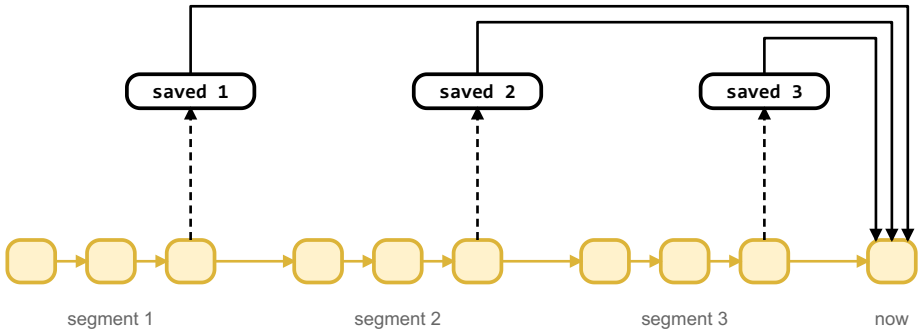
Last chapter's attention was a way around that. Instead of trusting one hidden state to have remembered everything, each prediction looks back at every earlier hidden state directly and decides how much each one matters. Pictured below is what that does to the distance between words:



With attention **every word is a single step from every other**, for the forward pass and for the gradient. The price is that every word gets compared with every word before it, 1,275 comparisons for our 50 words where the RNN takes 50 steps, and about 5 billion for a 100,000-word book. The work grows with the square of the length, an RNN's only with the length.

Memory Caching

That price is why RNNs haven't quite gone away. In February 2026 a group at Google Research (Behrouz and others) proposed **Memory Caching**, an RNN that reads its text in segments and saves a copy of its hidden state at the end of each one, so later words can look back at the saved copies as well as their own hidden state:



Its memory grows by one hidden state per segment rather than one per word, a middle ground between an RNN's fixed memory and attention's growing one. It generally improved the recurrent models they added it to, and closed much of the gap to attention-based models on recall tasks.

The models behind today's chatbots drop the recurrence altogether, though, and build the whole network out of attention, with no hidden state carried from word to word at all.