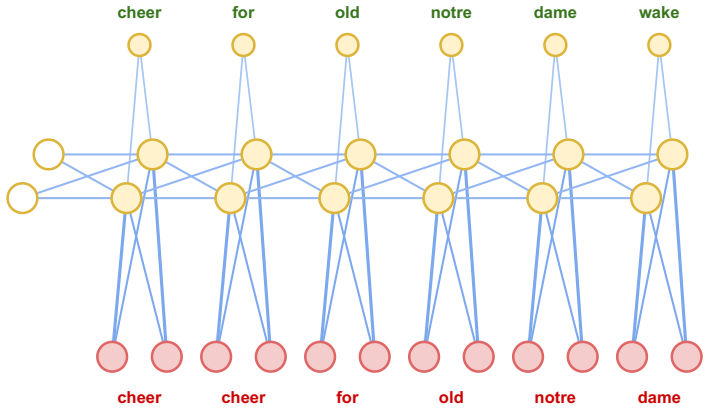


UNIT 03 · CHAPTER 3

Recurrent Neural Networks

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

Sequence Data

Our convolutional network looked at one `32 × 32` photo at a time and said what was in it. Each photo was classified completely on its own, so if we shuffle the `10,000` unseen photos every one of them still gets the same answer.



Text is different. Pictured above, under the photos, is the first line of the Notre Dame Victory March, one word to a box. Shuffled, it becomes **notre** **cheer** **for** **cheer** **dame** **old**, which is no longer the fight song. **The words only mean something in order**, and what comes next depends on what came before (after **notre** comes **dame**, in pretty much any text about this university).

Data like this, where the order matters and each piece has to be read in the context of the pieces before it, is called **sequence data**. Text is the obvious example, but speech, music, stock prices and video are all sequences too. As we saw at the end of the last chapter, a CNN can't take one in. It expects a grid of a fixed size, and a sentence can be any number of words long.

Tokenization

Unfortunately, a network only understands numbers. Our photos came as numbers already (`0` to `255` for each color of each pixel), but a fight song comes as letters, commas and spaces. Turning language into numbers a computer can use is the central problem of **Natural Language Processing** (NLP).

The first step is to cut the text into pieces called **tokens**, which is called **tokenization**. We'll keep it simple: lowercase everything, turn anything that isn't a letter into a space, and split on the spaces. We'll use the first four lines of the Victory March all the way through this chapter.

```python
import re
text = """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."""
clean = re.sub(r"[^a-z ]", " ", text.lower())
tokens = clean.split()
vocab = list(dict.fromkeys(tokens))  # first appearance
ids = [vocab.index(t) for t in tokens]
print(len(tokens), "tokens,", len(vocab), "words")
print(tokens[:9])
print(ids[:9])
```

OUTPUT

```
26 tokens, 22 words
['cheer', 'cheer', 'for', 'old', 'notre', 'dame', 'wake', 'up', 'the']
[0, 0, 1, 2, 3, 4, 5, 6, 7]
```

Tokenization Levels

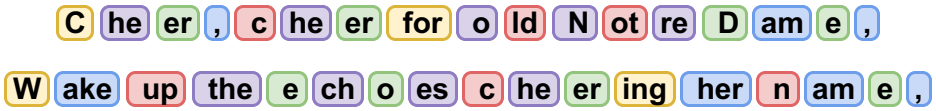
Our code does **word-level** tokenization, one token per word, and it isn't the only choice. Here is how our four lines come out at each level:

level	one token is	tokens	vocabulary
character-level	a character	136	25
word-level	a word	26	22
subword-level	a word or a piece of one	70	512
byte-level	a byte of the stored text	136	256

Our 22-word vocabulary only works because our text is four lines long. Real English needs hundreds of thousands of words, and a word the vocabulary has never seen (a typo, say) can't be tokenized at all. Character-level needs only a few dozen symbols, but our four lines become 136 tokens, each meaning very little on its own. **The goal is to keep the vocabulary small while keeping the sequences short**.

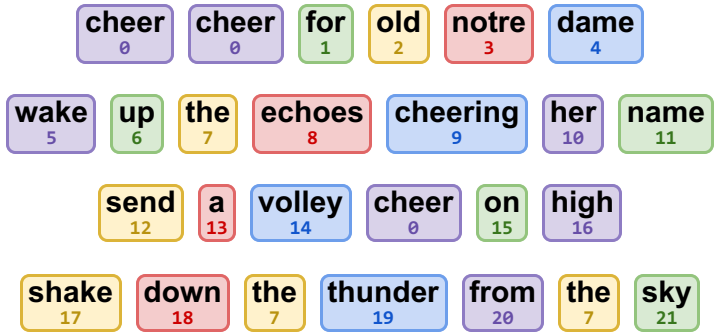
Subword Tokenizers

A **subword** tokenizer gives common words a token of their own and builds rarer ones out of pieces, which keeps both the vocabulary and the sequences reasonably small, and nearly every modern language model uses one. Below is how one splits our first two lines, the tokenizer of irishQwen (the small language model built in CSE `10124`), whose `512` tokens were made from Shakespeare.



With only `512` tokens to go around, just the most common words (**for** **up** **the** **her** **on** **from**) get one of their own. Even "cheer" is three pieces, **c** **he** **er**, and "cheering" is the same three plus **ing**. Tokenizers for real language models have `100,000` or more.

Token IDs

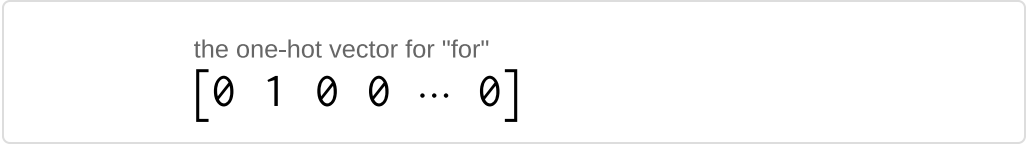


Above are all 26 of our word-level tokens. There are only 22 different words, since **cheer** and **the** each turn up three times. The list of different tokens is our **vocabulary**, and a token's position in it (in order of first appearance) is its **token ID**, written under each tile.

So now every token is a number. Are we actually gaining anything from this? Not much, yet. The IDs are just positions in a list, so if we handed them to a network as they are, it would decide that **thunder** (19) is much more like **sky** (21) than like **cheer** (0), and that **the** is worth seven **for**s. **The numbers say nothing about the words**.

One-Hot Encoding

The usual fix is a **one-hot** vector, a list of `22` zeros with a single `1` in the token's position. **for** (ID `1`) becomes



Now no word is bigger than any other. Unfortunately, every pair of words is also exactly the same distance apart (√2), so **cheer** is as close to **cheering** as it is to **sky**, and **a one-hot vector says which word it is and nothing else**. It doesn't scale well either. A vocabulary of `100,000` words would need vectors `100,000` long, all but one of them zeros.

What we'd like is a short list of numbers for each word, where words that are used in similar ways get similar numbers.

Word Embeddings

Much like our CNN turned each photo into a list of `4,096` learned features, in NLP we turn each token into a short vector called a **word embedding**, built from the only evidence a computer has about a word, the words around it.

Learning good embeddings takes billions of words, far more than our four lines, so we can borrow some. **GloVe** (Pennington, Socher and Manning, Stanford, `2014`) counted how often every pair of words turns up near each other in `6` billion words of Wikipedia and news, and learned `300` numbers for each of `400,000` words so that **words used in similar ways get similar vectors**. We only need the rows for our `22` words.

```
import torch
vocab, rows = [], []
for line in open("glove-victory-march.txt"):
    word, *nums = line.split()  # a word, 300 numbers
    vocab.append(word)
    rows.append([float(n) for n in nums])
E = torch.tensor(rows)  # our 22 words, in song order
print(E.shape)
```

OUTPUT

```
torch.Size([22, 300])
```

Cosine Similarity

To compare two words we scale both vectors to length `1` and take their dot product, the **cosine similarity** (`1` means they point the same way, `0` means unrelated). One matrix product gives us every pair at once, and we can ask which of our `22` words sits closest to each.

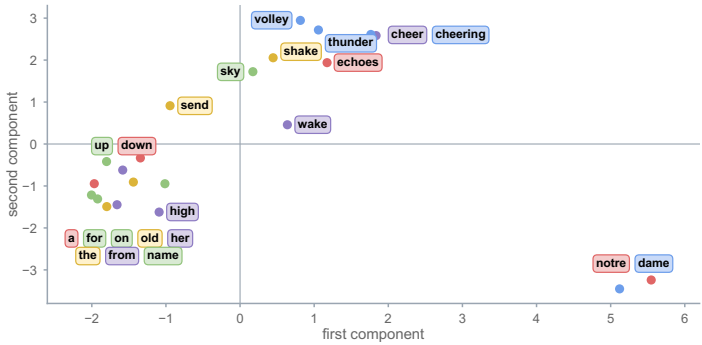
```
E = E / E.norm(dim=1, keepdim=True)  # length 1
S = E @ E.T  # cosine similarity of every pair
S = S - 2 * torch.eye(22)  # not its own neighbor
for w in ["cheer", "notre", "up", "thunder"]:
    s = S[vocab.index(w)]
    best = vocab[s.argmax()]
    print(f"{w!r:9} {best!r:10} {s.max():.2f}")
```

OUTPUT

```
'cheer'   'cheering' 0.61
'notre'   'dame'     0.91
'up'      'down'     0.75
'thunder' 'sky'      0.34
```

cheer finds **cheering** and **notre** finds **dame**, which is about what we'd hope. **thunder** and **sky** are a weaker match (`0.34`), and **up** and **down** are the second closest pair in the whole song, which is worth a closer look.

Embedding Space

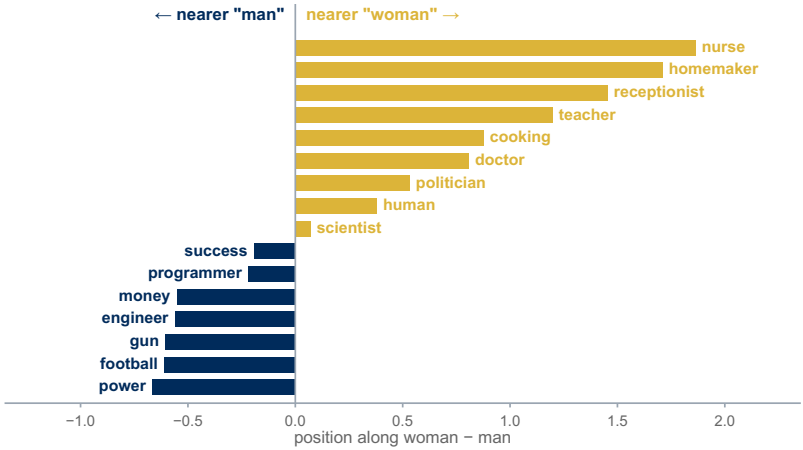


Pictured above is every word in our vocabulary, squashed from `300` numbers to `2` with PCA so we can look at it (only `28`% of the spread survives, so this is extremely approximate). **notre** and **dame** sit together, far from everything else, **cheer** and **cheering** are almost on top of each other, and the small words (a, for, on, the) bunch up on the left. Once tokens are points in space, we can compare words by how close they are.

Notice that **up** and **down** mean opposite things, yet they land right next to each other (`0.75`). Both turn up after the same kinds of words (**wake** **up**, **shake** **down**), and **related only ever means "used next to the same words"**. Embeddings learned from billions of words pick up whatever patterns the text has.

Bias in Embeddings

Unfortunately, the patterns include the text's biases. If we take the direction from "man" to "woman" in GloVe (the vector for "woman" minus the vector for "man") and see how far along it other words sit, jobs and ideas line up the way the articles talked about them. "nurse", "homemaker" and "receptionist" land far toward "woman", "engineer", "programmer" and "power" toward "man".

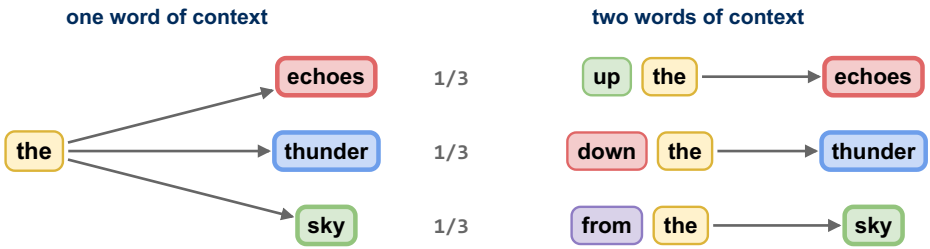


None of this was put there on purpose, the embedding only counted which words were used together. Embeddings learned from Google News articles went as far as answering "man is to computer programmer as woman is to ___" with "homemaker" (Bolukbasi and colleagues, `2016`, who named their paper after it). **A model built on top of embeddings inherits whatever they learned**.

Context

Now that words are numbers, let's give a network a job, predicting the next word. A feed-forward network takes one fixed-size input, so the natural thing to give it is the current word's embedding. Given **notre** it would learn to say **dame**, since in our song **dame** is the only word that ever follows **notre**.

Unfortunately, **the** is followed by **echoes** once, **thunder** once and **sky** once. A network that only sees **the** gets exactly the same input all three times, so **the best it can ever do is guess** (a third each). One more word back would settle it: **up** **the** is followed by **echoes**, **down** **the** by **thunder** and **from** **the** by **sky**.



We could feed it the last two words instead, but some other text will need three, or ten, and sentences don't come in a fixed length. The words that came before a word are its **context**, and what we want is a network that reads one word at a time and carries a summary of that context along with it.

Recurrent Neural Networks

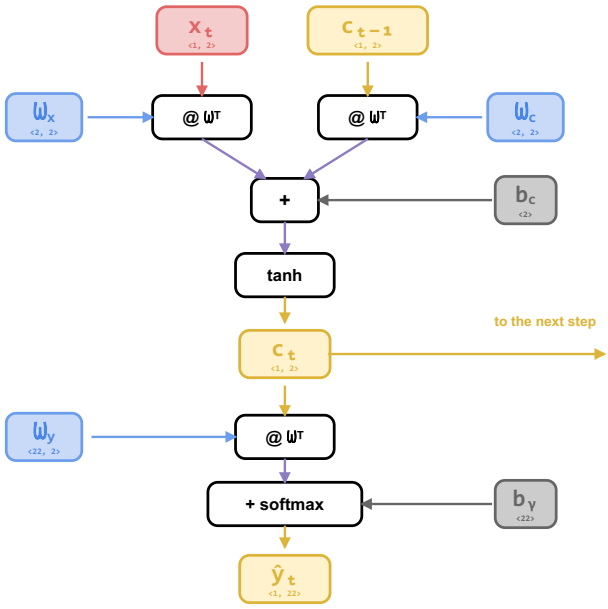
A **recurrent neural network** (RNN) does exactly that. At step t it takes the word's embedding x_t and the numbers it produced at the step before, c_{t-1}, called the hidden state, and combines them into a new hidden state:

[3.1] the new hidden state

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

W_x weights the new word, W_c weights the old hidden state and b_c is a bias, exactly like a fully connected layer except for the middle term. The hidden state starts out as all zeros, $c_0 = 0$. Because each new hidden state is built from the last one, **the hidden state after a word depends on every word before it**.

The RNN Cell

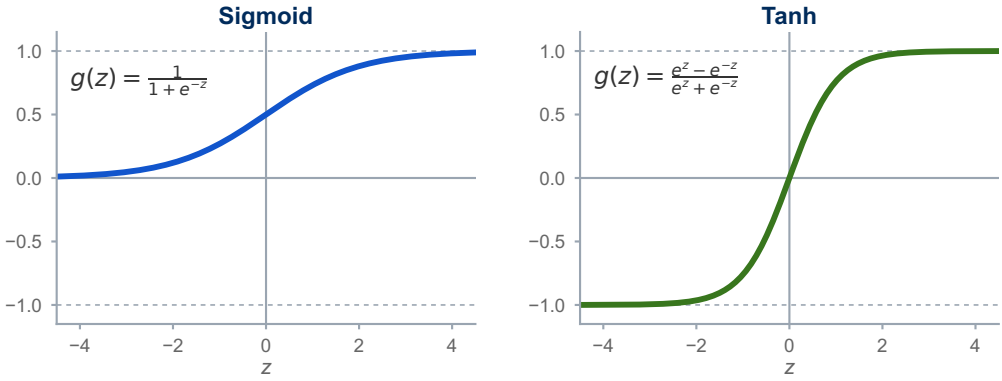


Pictured above is one step of our RNN, drawn as a computation graph in the same colors as our earlier networks. The top half is the equation on the last page, and the hidden state it makes goes on to the next step. The bottom half turns the hidden state into a prediction, which we'll come back to.

The idea goes back to at least **1986** (Michael Irwin Jordan, no relation), and Jeffrey Elman's **1990** network fed back its hidden layer, as ours does. One step like this is usually called a **cell**, since the same one is used again for every word.

The Tanh Function

The function wrapped around all of this is **tanh** (the hyperbolic tangent), which we saw next to the sigmoid and ReLU with feed-forward networks. Like the sigmoid it's S-shaped, but it runs from `-1` to `1` instead of `0` to `1`, and it gives `0` for an input of `0`.

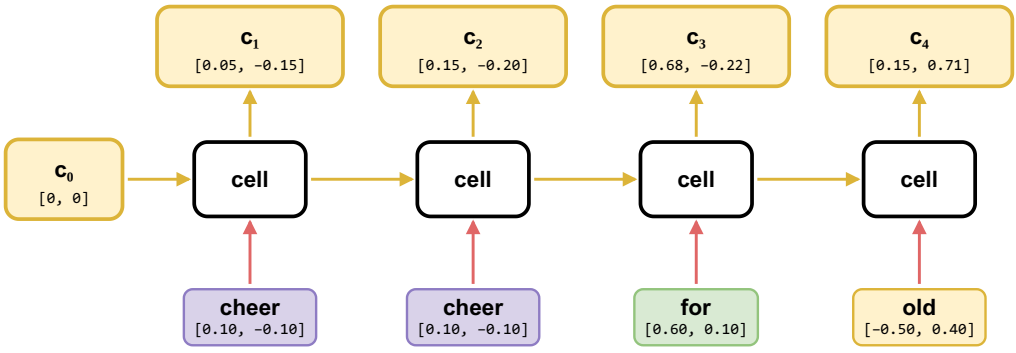


An RNN feeds its hidden state back into itself once per word, `26` times for our four lines. If each pass through W_c made the hidden state a little bigger, after `26` words it would be enormous. Tanh **keeps every number in the hidden state between −1 and 1** no matter how long the text is. Near `0` it's almost a straight line (tanh(`0.5`) = `0.462`) and past `±2` it's almost flat (tanh(`2`) = `0.964`).

Being centered on zero matters too. A sigmoid's outputs are always positive, while a tanh hidden state can push the next one up or down equally easily.

Unrolling

Because the hidden state loops back into the same cell, an RNN is usually drawn **unrolled**, with one copy of the cell for each word, side by side. Pictured below are our first four words.



It's important to remember that this isn't four different layers. **Every box uses the same weights**, so an RNN that reads 26 words has exactly as many weights as one that reads 4. The numbers under each word are its embedding (from a random table, since our network hasn't been trained) and the numbers above each cell are the hidden state it hands to the next step.

Notice that **cheer** goes in twice with the same embedding and comes out with two different hidden states.

An RNN by Hand

Let's work out those first two steps ourselves. Our embeddings and hidden state have two numbers each, the embedding table is random (rounded to one decimal), and we picked the weights by hand to keep the arithmetic simple, with the bias at `0`:

$$W_x = \begin{bmatrix} 1 & 0.5 \\ -0.5 & 1 \end{bmatrix} \qquad W_c = \begin{bmatrix} 0.5 & -0.5 \\ 0.5 & 0.5 \end{bmatrix}$$

The first word is **cheer**, with embedding `[0.1, −0.1]`. There's no hidden state yet (c_0 = `[0, 0]`), so only the first term counts:

[3.2]

step 1: cheer

$$c_1 = \tanh([0.1 \;\; -0.1]W_x^\top + [0 \;\; 0]W_c^\top)$$
$$= \tanh[0.05 \;\; -0.15] = [0.0500 \;\; -0.1489]$$

The second **cheer** has the same embedding, so the first term is the same `[0.05, −0.15]`, but now the hidden state adds $c_1 W_c^\top$ = `[0.0994, −0.0495]`:

[3.3]

step 2: cheer again

$$c_2 = \tanh[0.05 + 0.0994 \;\; -0.15 - 0.0495]$$
$$= \tanh[0.1494 \;\; -0.1995] = [0.1483 \;\; -0.1969]$$

Same word, same weights, different hidden state. **The second cheer knows it came after a cheer.**

The Forward Pass

Running every word through the cell in order is the RNN's forward pass, a loop in PyTorch. Below we print the hidden state at each **the**:

```python
import re
import numpy as np
import torch
text = """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."""
tokens = re.sub(r"[^a-z ]", " ", text.lower()).split()
vocab = list(dict.fromkeys(tokens))
ids = [vocab.index(t) for t in tokens]
rng = np.random.default_rng(0)
E = torch.tensor(rng.normal(size=(22, 2)).round(1),
                 dtype=torch.float32)  # embeddings
W_x = torch.tensor([[1.0, 0.5], [-0.5, 1.0]])
W_c = torch.tensor([[0.5, -0.5], [0.5, 0.5]])
b_c, c = torch.zeros(2), torch.zeros(2)  # bias, c₀
for t, i in enumerate(ids):
    c = torch.tanh(E[i] @ W_x.T + c @ W_c.T + b_c)
    if tokens[t] == "the":
        print(tokens[t-2:t+1], c.round(decimals=3))
```

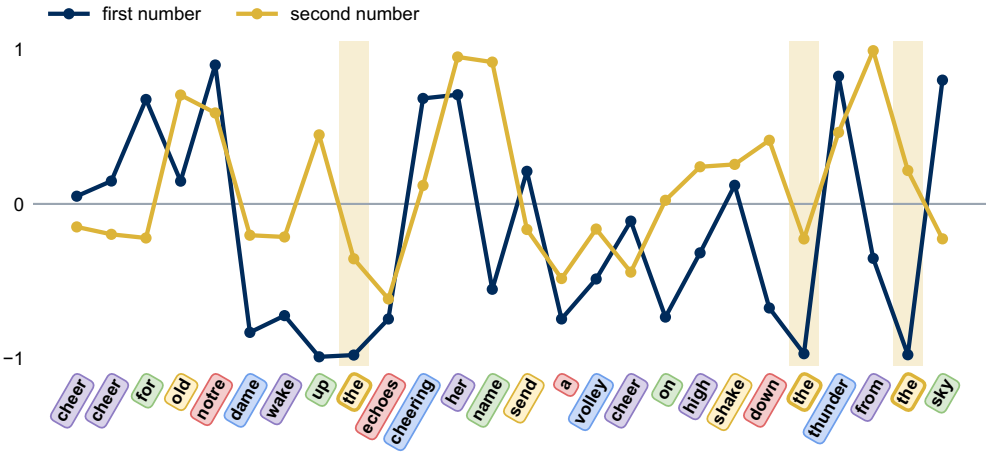
OUTPUT

```
[wake, up, the] tensor([-0.9790, -0.3550])
[shake, down, the] tensor([-0.9700, -0.2270])
[thunder, from, the] tensor([-0.9770,  0.2170])
```

Three times the network reads **the**, with the same embedding, and it ends up somewhere different each time (mostly in the second number, `-0.355`, `-0.227` and `0.217`). So **a layer reading this hidden state can tell the three apart**, which a network that only saw **the** never could.

Hidden States

Pictured below is the hidden state after every one of our 26 words, its first number in navy and its second in gold, with the three **the**s shaded.



It jumps around a lot, since the weights are made up and nothing has asked it to mean anything yet. Notice, though, what happens at the three **the**s. Each one pushes the first number down near −1, and the words before it decide the second.

That's the whole job of a hidden state, **a summary of everything read so far**, small enough to carry from one word to the next. Two numbers can't summarize much. Real RNNs use hundreds, and training decides what they hold on to.

RNNs in PyTorch

PyTorch has both pieces built in. `nn.Embedding` is the embedding table, one row per token ID, and `nn.RNN` runs the whole loop. Below we give them our weights instead of their own random ones and read the four lines again:

```
import re
import numpy as np
import torch
from torch import nn
text = """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."""
tokens = re.sub(r"[^a-z ]", " ", text.lower()).split()
vocab = list(dict.fromkeys(tokens))
ids = [vocab.index(t) for t in tokens]
E = np.random.default_rng(0).normal(size=(22, 2))
W_x = torch.tensor([[1.0, 0.5], [-0.5, 1.0]])
W_c = torch.tensor([[0.5, -0.5], [0.5, 0.5]])
emb = nn.Embedding(22, 2)
rnn = nn.RNN(2, 2, batch_first=True)
with torch.no_grad():  # our weights, not random ones
    emb.weight.copy_(torch.tensor(E.round(1)))
    rnn.weight_ih_l0.copy_(W_x)
    rnn.weight_hh_l0.copy_(W_c)
    rnn.bias_ih_l0.zero_()
    rnn.bias_hh_l0.zero_()
    x = emb(torch.tensor([ids]))  # 1 song, 26 words
    out, c = rnn(x)
print(tuple(x.shape), tuple(out.shape), tuple(c.shape))
print(out[0, 8].round(decimals=3))  # wake up the
```

OUTPUT

```
(1, 26, 2) (1, 26, 2) (1, 1, 2)
tensor([-0.9790, -0.3550])
```

Inside nn.RNN

On the last page, `out` held the hidden state after every word (what we need to predict every next word) and `c` only the last one. The weights that made them are stored under names of their own:

```
from torch import nn
rnn = nn.RNN(2, 2, batch_first=True)
for name, p in rnn.named_parameters():
    print(name, tuple(p.shape))
```

OUTPUT

```
weight_ih_l0 (2, 2)
weight_hh_l0 (2, 2)
bias_ih_l0 (2,)
bias_hh_l0 (2,)
```

`weight_ih_l0` ("input to hidden", layer `0`) is our W_x and `weight_hh_l0` ("hidden to hidden") is our W_c, which is why the hidden state after **wake** **up** **the** on the last page matched our loop. PyTorch keeps two bias vectors and adds them together, so its b_c is `bias_ih_l0 + bias_hh_l0`, and our cell's `10` numbers are `12` in PyTorch.

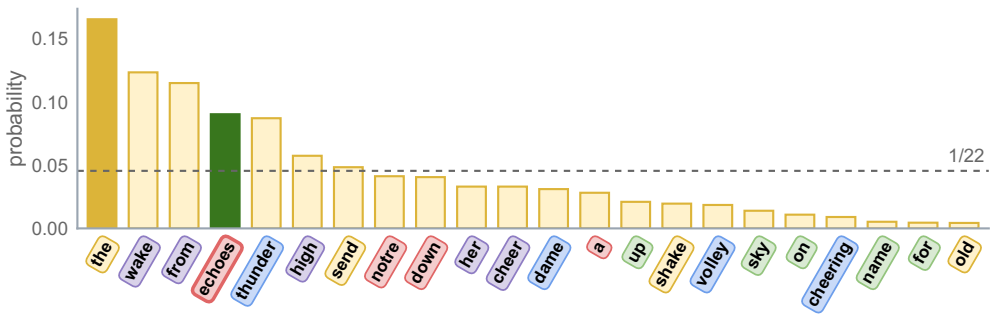
Outputs

To predict the next word, each hidden state goes through one more layer, with a softmax to turn its scores into one probability per word in the vocabulary (as our CNN did for its ten kinds of photo):

[3.4] the prediction at step t

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

For us W_y is 22×2, one row per word. Pictured below is what our network says after **wake** **up** **the**, with W_y random as well and $b_y = 0$.



It thinks the next word is most likely **the** (0.165) and gives **echoes** 0.090. Averaged over the 25 words that have a next word, it gives the right one a probability of 0.051, which is **barely better than picking at random** (1/22 = 0.045). That's what made-up weights get us.

An RNN From Scratch

In Backpropagation we wrote each layer of a network as a small class with a `forward` method, and in CNNs we added a convolutional layer and pooling the same way. An RNN needs two new layers. The first is the **embedding layer**:

```
import torch

class Embedding:
    def __init__(self, n_tokens, d):
        self.E = torch.randn(n_tokens, d) * 0.1

    def forward(self, ids):  # (B, T) token IDs
        self.ids = ids       # keep for backward
        return self.E[ids]   # (B, T, d): one row per ID
```

It holds the embedding table, one row per token in the vocabulary, and its `forward` is **a lookup**: a batch of token IDs, shape `(B, T)`, becomes a batch of vectors, shape `(B, T, d)`. That is the same as multiplying each one-hot vector by the table (One-Hot Encoding), without building the one-hot vectors.

The Recurrent Layer

The second is the **recurrent layer**, our cell run over the words in order:

```
class RNNLayer:
    def __init__(self, n_in, n_h):
        self.Wx = torch.randn(n_h, n_in) / n_in**0.5
        self.Wc = torch.randn(n_h, n_h) / n_h**0.5
        self.b = torch.zeros(n_h)

    def step(self, x_t, c_prev):  # one word, c_t
        return torch.tanh(x_t @ self.Wx.T
                          + c_prev @ self.Wc.T + self.b)

    def forward(self, X):  # (B, T, n_in)
        B, T, _ = X.shape
        c = torch.zeros(B, self.Wc.shape[0])  # c_0
        self.states = [c]  # keep every c_t for BPTT
        for t in range(T):  # one word at a time
            c = self.step(X[:, t], c)  # same weights
            self.states.append(c)
        return torch.stack(self.states[1:], dim=1)
```

The `step` method is the equation from The RNN Cell, for a whole batch of sequences at once. The `forward` method is our loop from The Forward Pass: it starts from a hidden state of zeros and calls the same `step`, with the same `Wx` and `Wc`, for every word. It also keeps every hidden state in `self.states`, since training will need them in the next chapter.

The Rest of the Network

The output layer from the last page is a linear layer, and we already have one. `LinearLayer` and `Network` are the classes from Backpropagation, with only their `forward` methods for now:

```
class LinearLayer:
    def __init__(self, n_in, n_out):
        self.W = torch.randn(n_out, n_in) * (2/n_in)**0.5
        self.b = torch.zeros(n_out)

    def forward(self, X):  # every c_t at once
        self.X = X         # keep for backward
        return X @ self.W.T + self.b

class Network:
    def __init__(self, layers):
        self.layers = layers

    def forward(self, X):
        for layer in self.layers:
            X = layer.forward(X)
        return X  # logits
```

The recurrent layer hands on a hidden state for every word, shape `(B, T, n_h)`, and `@` multiplies the last dimension, so the linear layer scores **every word's next word at once**. `Network` doesn't care what kind of layers it holds.

Running Our RNN From Scratch

Below we put the four lines through an embedding layer, a recurrent layer with two hidden numbers and a linear layer back out to our 22 words, all with random weights:

```
import re
text = """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."""
tokens = re.sub(r"[^a-z ]", " ", text.lower()).split()
vocab = list(dict.fromkeys(tokens))
X = torch.tensor([[vocab.index(t) for t in tokens]])
torch.manual_seed(0)
model = Network([Embedding(22, 2), RNNLayer(2, 2),
                 LinearLayer(2, 22)])
logits = model.forward(X)
emb, rnn, out = model.layers
print("IDs", tuple(X.shape), "-> logits",
      tuple(logits.shape))
print("hidden states:", len(rnn.states))
n = [emb.E, rnn.Wx, rnn.Wc, rnn.b, out.W, out.b]
print("weights:", sum(w.numel() for w in n))
```

OUTPUT

```
IDs (1, 26) -> logits (1, 26, 22)
hidden states: 27
weights: 120
```

One song of 26 words comes out as 26 rows of 22 scores, one for each next word, and the recurrent layer kept 27 hidden states. The network has 120 weights: 44 in the embedding table, 10 in the cell and 66 in the output layer. **None of them depend on how long the song is**.

Our Weights, From Scratch

To check our classes, we can give them the weights from the earlier pages (the same seed for the embedding table and W_y, our W_x and W_c, and zero biases) and run the song again:

```
import numpy as np
rng = np.random.default_rng(0)  # earlier slides' weights
E = rng.normal(size=(22, 2)).round(1)
W_y = rng.normal(size=(22, 2)).round(1)
emb.E = torch.tensor(E, dtype=torch.float32)
rnn.Wx = torch.tensor([[1.0, 0.5], [-0.5, 1.0]])
rnn.Wc = torch.tensor([[0.5, -0.5], [0.5, 0.5]])
rnn.b = torch.zeros(2)
out.W = torch.tensor(W_y, dtype=torch.float32)
out.b = torch.zeros(22)

p = torch.softmax(model.forward(X), dim=-1)
c = rnn.states[9]  # after "wake up the"
print("c:", c.round(decimals=3))
echoes = p[0, 8, vocab.index("echoes")]
print("p('echoes'):", echoes.round(decimals=3))
```

OUTPUT

```
c: tensor([[-0.9790, -0.3550]])
p(echoes): tensor(0.0900)
```

After wake up the we get the hidden state from The Forward Pass and the `0.090` for echoes from Outputs, so **our classes compute the same network** as the loop.

Checking Against PyTorch

We can check against PyTorch the same way we did for the loop, by copying our weights into `nn.Embedding`, `nn.RNN` and `nn.Linear`:

```
from torch import nn
emb_t = nn.Embedding(22, 2)
rnn_t = nn.RNN(2, 2, batch_first=True)
out_t = nn.Linear(2, 22)
with torch.no_grad():  # copy our weights in
    emb_t.weight.copy_(emb.E)
    rnn_t.weight_ih_l0.copy_(rnn.Wx)
    rnn_t.weight_hh_l0.copy_(rnn.Wc)
    rnn_t.bias_ih_l0.copy_(rnn.b)
    rnn_t.bias_hh_l0.zero_()  # PyTorch has two biases
    out_t.weight.copy_(out.W)
    out_t.bias.copy_(out.b)
    hidden, _ = rnn_t(emb_t(X))
    logits_t = out_t(hidden)
diff = (logits_t - model.forward(X)).abs().max()
print("largest difference:", diff)
```

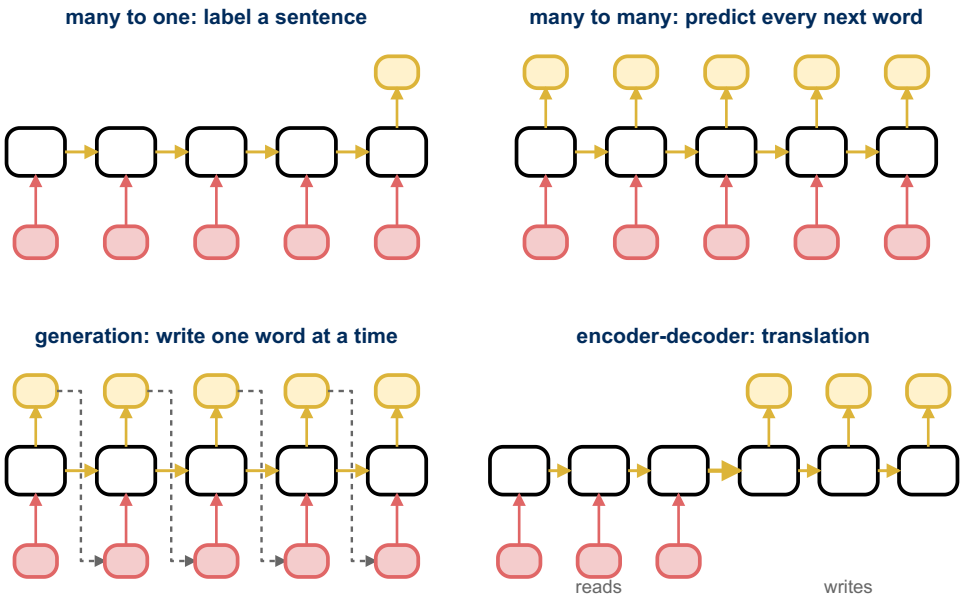
OUTPUT

```
largest difference: tensor(0.)
```

The scores are identical. PyTorch keeps a second bias in the cell (Inside nn.RNN), which we set to zero, and otherwise **`nn.RNN` is our loop**, written in faster code. When we train an RNN in the next chapter, our classes will need a `backward` method, while PyTorch will work the gradients out for us.

Uses of RNNs

Our network gives an output after every word, a guess at the next one. That isn't the only way to wire an RNN up, and which outputs we keep (and what we feed in) decides what it can be used for.



Pictured above, **many to one** reads a whole sequence and only answers at the end, using the last hidden state, which is how an RNN labels a sentence (is this review positive?). **Many to many** answers after every word, like our next-word predictions. The bottom two are the uses that made RNNs famous, writing text and translating it.

Text Generation

A network that predicts the next word can also write. We give it a first word, take the word it thinks is likeliest, and feed that back in as the next input, one word at a time:

```
import re
import numpy as np
import torch
text = """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."""
tokens = re.sub(r"[^a-z ]", " ", text.lower()).split()
vocab = list(dict.fromkeys(tokens))
rng = np.random.default_rng(0)
E = torch.tensor(rng.normal(size=(22, 2)).round(1),
                 dtype=torch.float32)  # embeddings
W_y = torch.tensor(rng.normal(size=(22, 2)).round(1),
                   dtype=torch.float32)  # output layer
W_x = torch.tensor([[1.0, 0.5], [-0.5, 1.0]])
W_c = torch.tensor([[0.5, -0.5], [0.5, 0.5]])
c = torch.zeros(2)
word, song = "cheer", ["cheer"]
for t in range(9):
    i = vocab.index(word)
    c = torch.tanh(E[i] @ W_x.T + c @ W_c.T)
    p = torch.softmax(c @ W_y.T, dim=0)
    word = vocab[p.argmax()]  # the likeliest next word
    song.append(word)
print(song[:5])
print(song[5:])
```

OUTPUT

```
['cheer', 'send', 'on', 'shake', 'shake']
['for', 'old', 'shake', 'shake', 'for']
```

Not exactly the fight song, but that's what made-up weights write.

Sampling

Always taking the likeliest word tends to get stuck in loops (**shake** **shake** **for**), so in practice the next word is drawn at random from the probabilities instead, which is called **sampling**. How random is set by the **temperature** T, which divides the scores before the softmax:

T = 1 is the usual softmax

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

After **cheer**, the likeliest word for our network is **send**, at `0.064`. At T = `0.5` it gets `0.087` and at T = 2 only `0.054`, closer to `1/22` = `0.045`. **A low temperature sharpens the guesses toward the likeliest word, a high one flattens them toward picking at random**.

Trained on enough text, a generating RNN gets surprisingly good. Andrej Karpathy's `2015` RNNs, which read and wrote one character at a time, produced passable fake Shakespeare and Wikipedia articles.

Machine Translation

Translation is harder, since a sentence and its translation rarely have the same number of words, or put them in the same order. The answer, from two groups in `2014` (Sutskever, Vinyals and Le at Google, and Cho and colleagues in Montreal), was to use two RNNs. The first, the **encoder**, reads the whole sentence and produces no outputs at all. Its last hidden state is handed to the second, the **decoder**, as its starting hidden state, and the decoder generates the translation one word at a time, like the last page, until it predicts a special end-of-sentence token.

Everything the decoder knows about the original sentence is in that one hidden state, which is why it's often called the **context vector**. It's also called an **encoder-decoder** (or sequence-to-sequence) model, and with LSTMs in place of our plain cells, plus the attention of a few pages on, it was good enough that in `2016` Google Translate switched to one.

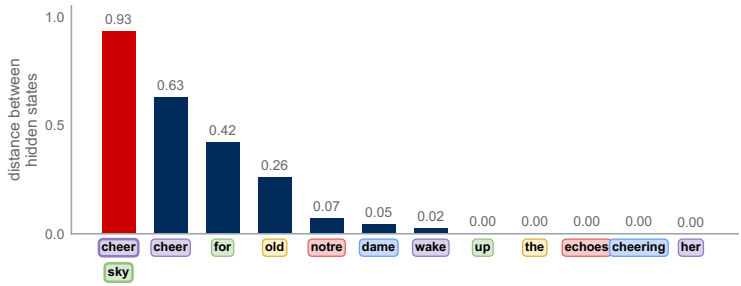
Pros and Cons of RNNs

An RNN reads a sequence of any length with one fixed set of weights. Ours has `120` numbers to learn (`44` in the embedding table, `10` in the cell and `66` in the output layer), whether it reads four words or four thousand. Its hidden state carries along the context a feed-forward network never sees, and it can give an output after every word, or only after the last one (to label a whole sentence, say).

Unfortunately, **it can only read one word at a time**. Each step needs the hidden state from the step before, so `26` words means `26` steps, one after another. Our CNN can label a thousand photos at once, because none of them needs the others' answers, but an RNN can't start on word `10` until it has finished word `9`. On long texts and big GPUs that makes RNNs slow to train.

Long-Term Dependencies

Worse, it forgets. Below we run the forward pass twice, on the song as written and with **sky** in place of the first **cheer**, and measure how far apart the two hidden states are after each word.



The gap is `0.93` after the first word, `0.26` by **old** and `0.02` by **wake**. By **the**, the ninth word, it's down to `0.001`, so **it effectively has no idea how the song started**. Each step mixes the old hidden state with a new word and squashes the result, and our W_c shrinks the old part by about `30%` every time.

A word that depends on another word far back is called a **long-term dependency**, and RNNs are bad at them. A trained network holds on for longer than our made-up one, but every RNN has this problem to some degree.

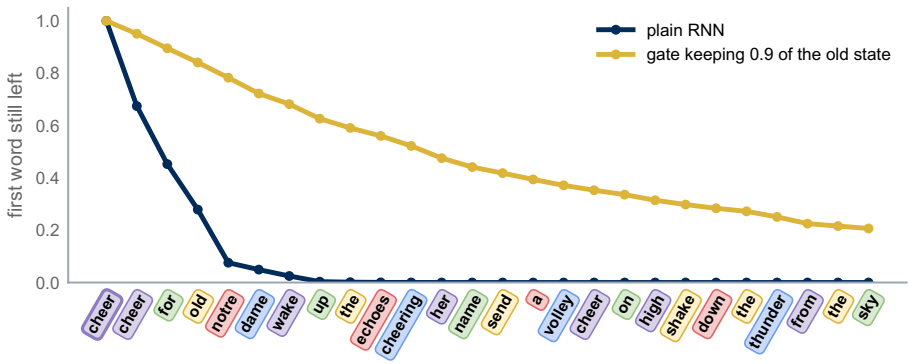
Gates

Our network forgets because every step replaces the hidden state with a mix of the new word and the old state, squashed. What if each step could choose how much of the old hidden state to keep? Write $\tilde{c}_t$ for the new hidden state our cell would make, and let a vector z_t of numbers between `0` and `1` blend the two:

keep z of the old state, take 1 − z of the new

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

A vector like z_t that multiplies something number by number ($\odot$) is called a **gate**. Pictured below is our forgetting experiment again, with a gate that always keeps `0.9` of the old state:



By the ninth word the plain RNN has `0.1%` of the first word left, the gated one `59%`, and at the end of the song still `21%`.

LSTMs and GRUs

Of course a gate stuck at `0.9` isn't much use, since it remembers everything just as hard, the words that matter and the ones that don't. In a real network the gate is a layer of its own, a sigmoid looking at the new word and the old hidden state, $z_t = \sigma(x_t\, W_{zx}^{\top} + c_{t-1}\, W_{zc}^{\top} + b_z)$, so the network learns when to hold on and when to let go.

That's (most of) a **Gated Recurrent Unit** (GRU, Cho and colleagues, `2014`), which adds a second gate deciding how much of the old state goes into $\tilde{c}_t$. The older **Long Short-Term Memory** (LSTM, Hochreiter and Schmidhuber, `1997`) uses three gates and keeps a second memory vector, the cell state, alongside the hidden state. Every gate is one more layer of weights:

```
from torch import nn
for cell in [nn.RNN(2, 2), nn.GRU(2, 2), nn.LSTM(2, 2)]:
    n = sum(p.numel() for p in cell.parameters())
    print(type(cell).__name__, n)
```

OUTPUT

```
RNN 12
GRU 36
LSTM 48
```

In PyTorch either one is a drop-in replacement for `nn.RNN`, three and four times its size. Why gates make these networks so much easier to train is a story about gradients, and the next chapter's.

The Bottleneck

Gates help an RNN hold on, but it still has to fit everything into one hidden state of a fixed size. For a translation that's the context vector, and every word of the original sentence has to survive in it until the decoder needs it, however long the sentence is. Ours has two numbers to carry the whole song.

This is called the **bottleneck** problem, and it showed. Cho and colleagues found in `2014` that their encoder-decoder's translations got steadily worse as the sentences got longer.

The encoder did make a hidden state after every word, though, and each one is a summary of the sentence up to that word. Rather than throwing all but the last one away, the decoder could keep them all and, at each word it writes, look back at whichever ones it needs.

Attention

That's what Bahdanau, Cho and Bengio did in **2014**, and they called it **attention**. Before writing each word, the decoder scores every one of the encoder's hidden states against its own current hidden state s (the simplest score is a dot product), turns the scores into weights with a softmax, and adds up the encoder's hidden states with those weights:

how well hidden state j matches

$$e_j = s \cdot c_j$$

weights that add up to 1

$$\alpha = \mathrm{softmax}(e)$$

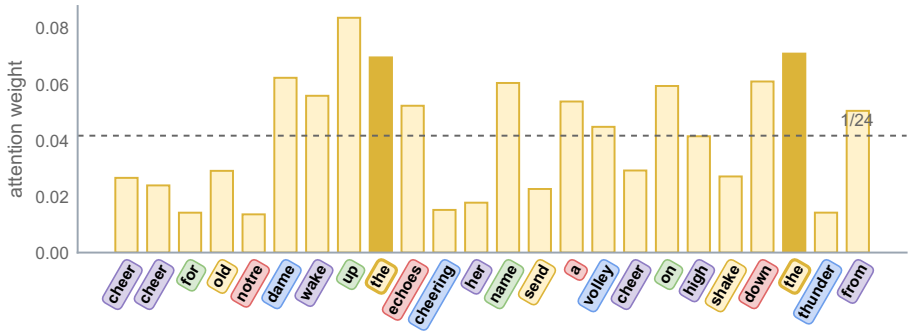
a context vector for this word

$$v = \sum_j \alpha_j c_j$$

So every word the decoder writes gets its own context vector, built mostly from the hidden states it gave the most weight. Translating into French, the weights for each French word mostly picked out the English word it came from, and the translations stopped falling apart on long sentences.

Attention Weights

We have no decoder, but we can let our network look back the same way. At the last **the** (word `25`) we score its hidden state against the hidden states after each of the `24` words before it:



The biggest weights go to **up** (`0.084`) and the other two **the**s (`0.071` and `0.070`), against `1/24` = `0.042` if every word counted the same, because their hidden states are the most like this one. Its context vector, `[-0.49, 0.08]`, is mostly those words.

Unfortunately, with made-up weights that's as far as we can get. Every weight, gate and score in this chapter needs to be learned from data, and every weight in an RNN is used at every step, so training one means following the loss back through every step of the loop, which is where we pick up next. (Attention will be back too, without the RNN at all.)