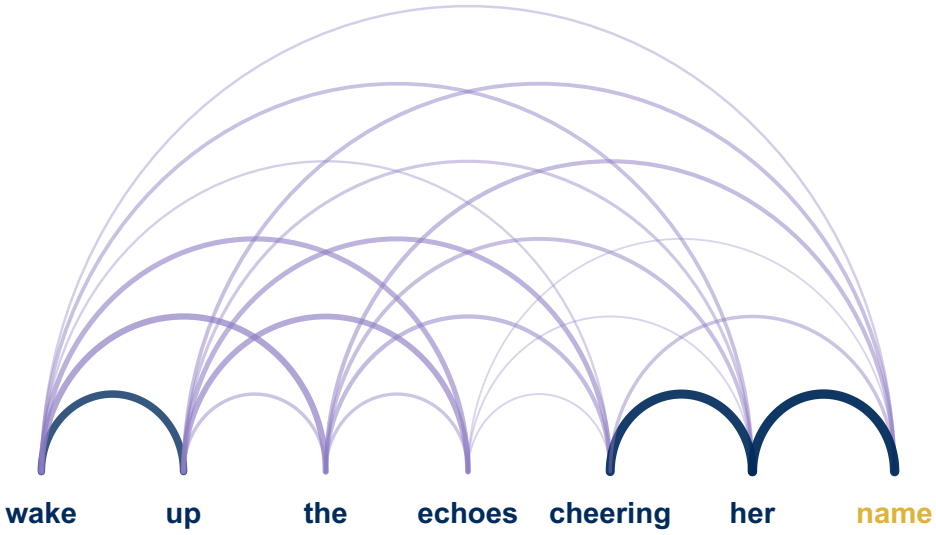


UNIT 03 · CHAPTER 5

Self-Attention

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

Tokens and Embeddings

Last chapter our **RNN** read the Victory March the way we do, one word at a time, carrying a hidden state from each word to the next. For this chapter one line of the chorus is enough, "wake up the echoes cheering her name".

Before a network can read anything, the text has to become numbers. As with our recurrent networks, every word is a token, so our line is `7` tokens. Each token is then looked up in a table of **embeddings**, one row of numbers per token. Real embeddings are long (GPT-3's have `12,288` numbers), ours have d = 4:

```python
import numpy as np
import torch

text = "wake up the echoes cheering her name"
tokens = text.split()  # word-level tokens
print(tokens)
rng = np.random.default_rng(28)
X = rng.uniform(-1, 1, (7, 4)).round(1) + 0.0
X = torch.tensor(X, dtype=torch.float32)
print(X)
```

OUTPUT

```
[wake, up, the, echoes, cheering, her, name]
tensor([[ 0.7000,  0.8000,  0.5000, -0.9000],
        [-0.9000, -0.3000,  0.8000,  0.9000],
        [ 0.1000,  0.6000, -0.3000,  0.6000],
        [ 0.6000, -0.9000,  0.9000, -0.3000],
        [-0.6000,  0.0000,  0.2000, -1.0000],
        [-0.7000, -0.3000, -0.6000,  0.2000],
        [-0.8000,  0.5000,  0.0000, -0.4000]])
```

Static Embeddings

Our embeddings are made up, drawn at random and rounded to one decimal so we can work everything by hand (a trained model learns its embeddings like any other weights). Each row of **X** is one token, from "wake" at the top to "name" at the bottom.

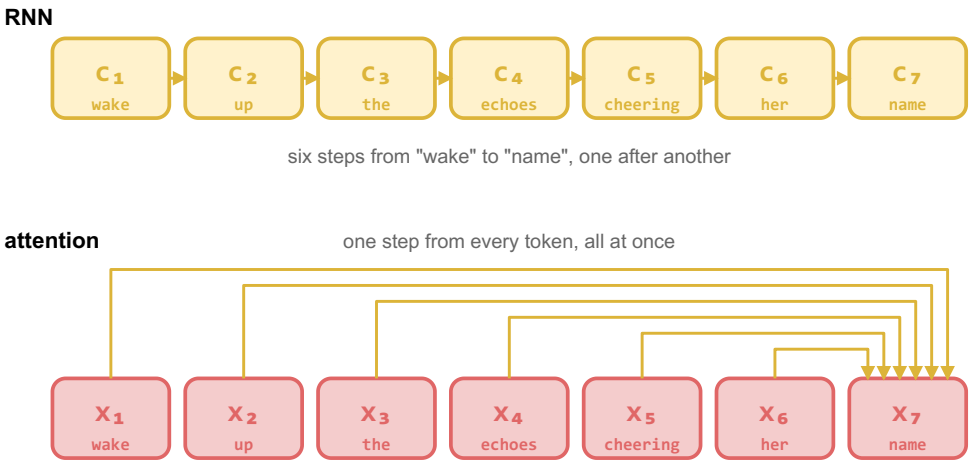
Notice that the table has one row per word and nothing else goes into it. "her" gets `[-0.7, -0.3, -0.6, 0.2]` in our line, where it means Notre Dame (who isn't even one of the tokens), and exactly the same row in any other sentence, whoever "her" is there. "echoes" is a noun in our line and a verb in "the stadium echoes", and it gets the same row in both.

As we saw last chapter, an embedding that doesn't change regardless of where the word appears or what other words are around it is a **static embedding**. Every embedding we've used so far has been static, so a token's numbers know nothing about its sentence.

The RNN's way around this is its hidden state. By the time it reaches "name", c_7 has seen all six tokens before it, so "name" is at least read in some context, even if its embedding isn't.

Sequential Processing

Unfortunately, the hidden state comes with two problems. The first is that the RNN has to process the sequence in order. c_7 needs c_6, which needs c_5, and so on back to "wake", so our `7` tokens take `7` steps that can't be run at the same time, however many processors we have. GPT-3 reads up to `2,048` tokens at once, which would be `2,048` steps in a row.



The second is that "wake" only reaches "name" through six updates of the hidden state, and each one squeezes it a little more. As we saw with vanishing gradients, **the RNN loses information from the earlier tokens** as the sequence goes on.

Pictured above at the bottom is the alternative: let "name" look at every token directly, all at once. That leaves the question of how much it should take from each one.

Similarity Scores

Some tokens should matter more to "name" than others ("her" says whose name, "the" says very little). The simplest measure of how related two embeddings are is their **dot product**. Two embeddings pointing the same way give a big positive number, two pointing opposite ways a negative one. For "name" and "her":

name · her, number by number

$$x_{\text{name}} \cdot x_{\text{her}} = (-0.8)(-0.7) + (0.5)(-0.3) + (0)(-0.6) + (-0.4)(0.2) = 0.33$$

Doing the same with every token gives "name" a score for each:

	wake	up	the	echoes	cheering	her	name
score	0.20	0.21	−0.02	−0.81	0.88	0.33	1.05

Unfortunately, **scores aren't weights**. They can be negative, they don't add up to anything in particular, and the next sentence would give a completely different total. What we want is a list of weights that are all positive and add up to `1`, so we can take a weighted average of the tokens.

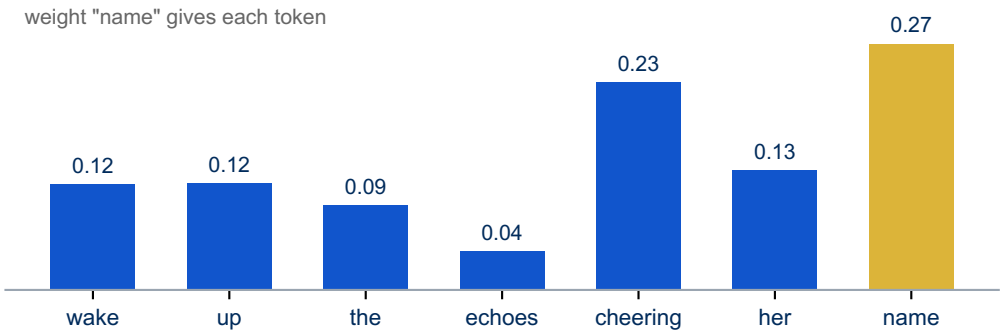
Softmax

Turning scores into weights is exactly what the **softmax** does, as we saw with the classifier outputs in Feed-Forward Networks. It raises e to the power of each score, which makes every one positive, then divides each by their total (`10.54` for "name"), which makes them add up to `1`:

	wake	up	the	echoes	cheering	her	name
score	0.20	0.21	−0.02	−0.81	0.88	0.33	1.05
e^{score}	1.22	1.23	0.98	0.44	2.41	1.39	2.86
weight	0.12	0.12	0.09	0.04	0.23	0.13	0.27

The order doesn't change, but **the gaps get stretched**, since a score `1` higher than another always gets $e \approx 2.72$ times the weight.

Attention



Pictured above are the weights "name" gives each token. Now we can take the weighted average of all seven embeddings, `0.12` of "wake" plus `0.12` of "up" and so on, which gives "name" a new vector, `[-0.44, 0.17, 0.13, -0.27]`. Change any word in the line and this vector changes too, so it isn't static.

Mixing in every token of the context, weighted by how relevant it is, is called **attention**. It's a mechanism through which we can focus on the parts of the context that matter most.

This version doesn't focus on much, though. "name" gives the most weight to itself (its score with itself, `1.05`, is the biggest), and "her" gets only `0.13`. Worse, **there's nothing here to learn**. Each embedding does three jobs at once (asking what's relevant, being checked for relevance and being passed along), and the embeddings alone fix the scores.

Queries, Keys and Values

In 2017 eight researchers, most of them at Google, published "Attention Is All You Need", which got rid of the RNN altogether and kept only attention. They describe it as a dictionary lookup, "mapping a query and a set of key-value pairs to an output".

So each token gets three vectors instead of one, its **query, key and value**, each from its own weight matrix. The query, $q = x\,W_Q$, is what it's looking for. The key, $k = x\,W_K$, is what it offers to be matched against. The value, $v = x\,W_V$, is what it passes along. Our made-up W_Q and the query it gives "name":

0.9	1.0	0.8	0.9
−0.8	−0.3	−0.7	−0.1
0.9	0.1	0.9	0.3
−0.7	0.9	0.3	0.1

the query for name

$$q_{\text{name}} = x_{\text{name}} W_Q = \begin{bmatrix} -0.8 & 0.5 & 0 & -0.4 \end{bmatrix} W_Q$$
$$= \begin{bmatrix} -0.84 & -1.31 & -1.11 & -0.81 \end{bmatrix}$$

A Python dictionary hands back the value whose key matches the query exactly. Attention is a softer version, **every key matches a little, and we get back a blend of every value**. W_Q, W_K and W_V are weights, so training learns them with backpropagation (ours are random).

Attention Scores

Now the score between "name" and each token is the dot product of the query for "name" with that token's key. For "her", whose key is $k_{her} = [0.83, -1.25, -1.38, -1.02]$:

name's query · her's key

$$q_{name} \cdot k_{her} = (-0.84)(0.83) + (-1.31)(-1.25) + (-1.11)(-1.38) + (-0.81)(-1.02)$$

$$= 3.30$$

All seven, as before:

	wake	up	the	echoes	cheering	her	name
$q_{name} \cdot k$	−3.71	−0.26	0.37	−2.94	0.06	3.30	0.74
divided by 2	−1.86	−0.13	0.19	−1.47	0.03	1.65	0.37

The second row divides every score by $\sqrt{d_k}$, the square root of the length of a key ($\sqrt{4} = 2$ for us). The longer the vectors, the bigger their dot products get. If every number in a query and a key is random with a standard deviation of 1, their dot product has a standard deviation of $\sqrt{d_k}$, and dividing by $\sqrt{d_k}$ brings it back to 1. Fed big scores, the softmax gives nearly everything to the biggest one and its gradients get extremely small. Even at $d_k = 4$, without the division "her" would get 0.83 of the weight instead of 0.51. So **we scale the scores to keep the softmax soft**.

The Weighted Sum of Values

A softmax of the scaled scores gives the weights, and the weighted sum of the values (not the embeddings, this time) is the new vector for "name":

token	weight	value v
wake	0.015	[1.09, 0.72, −0.68, −0.86]
up	0.086	[−0.26, −0.76, 1.36, −0.78]
the	0.119	[−0.20, 0.93, −0.03, 0.09]
echoes	0.023	[0.30, −0.51, 1.17, 0.09]
cheering	0.102	[0.54, −1.08, −0.92, −0.84]
her	0.513	[−0.59, −0.65, −0.33, 0.39]
name	0.142	[0.32, −0.52, −0.90, −0.95]

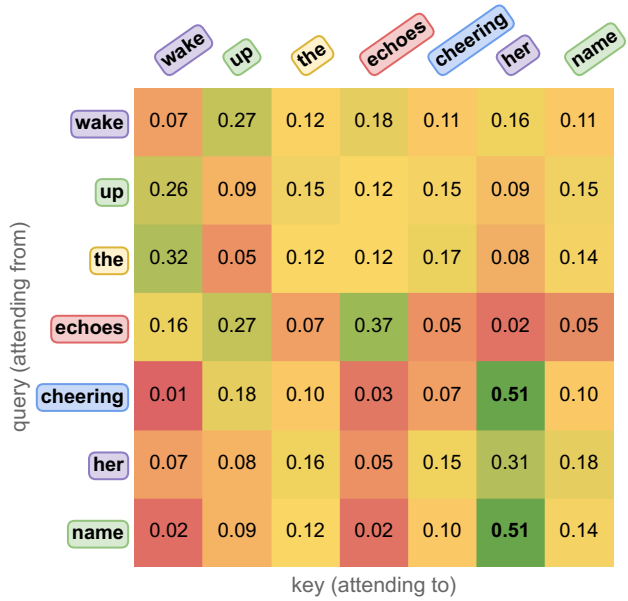
the output for name

$$z_{\text{name}} = 0.015\, v_{\text{wake}} + 0.086\, v_{\text{up}} + \cdots + 0.142\, v_{\text{name}}$$
$$= \begin{bmatrix} -0.22 & -0.47 & -0.26 & -0.09 \end{bmatrix}$$

This time "name" gives just over half its attention to "her", and only `0.14` to itself. Its new vector is mostly the value of "her", with a bit of "name", "the" and "cheering" mixed in. Notice that **this is luck**, since our weight matrices are random and nothing has learned that "her" is the word whose name it is.

Self-Attention

"name" took seven dot products, one with each key. Every other token does the same with its own query. Stacking the queries as the rows of Q and the keys as the rows of K, all 49 scores come out of one matrix product, $Q K^T$, with one row per token. A softmax along each row turns them into weights:



The bottom row is the one we worked out for "name". Every token here attends to the tokens of its own sequence, which is why this is called **self-attention**. And since no row has to wait for any other, the whole grid is calculated in parallel, one-to-rest attention for every token at once, the exact opposite of an RNN's 7 steps in a row.

The Attention Equation

In one line, this is the equation from "Attention Is All You Need":

scaled dot-product attention

[5.1] $$\mathrm{Attention}(Q, K, V) = \mathrm{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

$Q\,K^T$ scores every query against every key, $\sqrt{d_k}$ keeps the scores in a sensible range, the softmax turns each row into weights and V gets their weighted sum. In PyTorch:

```python
import numpy as np
import torch
rng = np.random.default_rng(28)
X = rng.uniform(-1, 1, (7, 4)).round(1) + 0.0
WQ, WK, WV = rng.uniform(-1, 1, (3, 4, 4)).round(1)
X, WQ, WK, WV = [torch.tensor(M, dtype=torch.float32)
                 for M in (X, WQ, WK, WV)]
def attention(Q, K, V):
    S = Q @ K.T / K.shape[1] ** 0.5
    E = torch.exp(S)
    A = E / E.sum(dim=1, keepdim=True)
    return A @ V, A
Z, A = attention(X @ WQ, X @ WK, X @ WV)
print(A[-1].round(decimals=2))  # the weights for name
print(Z[-1].round(decimals=2))  # the output for name
```

OUTPUT

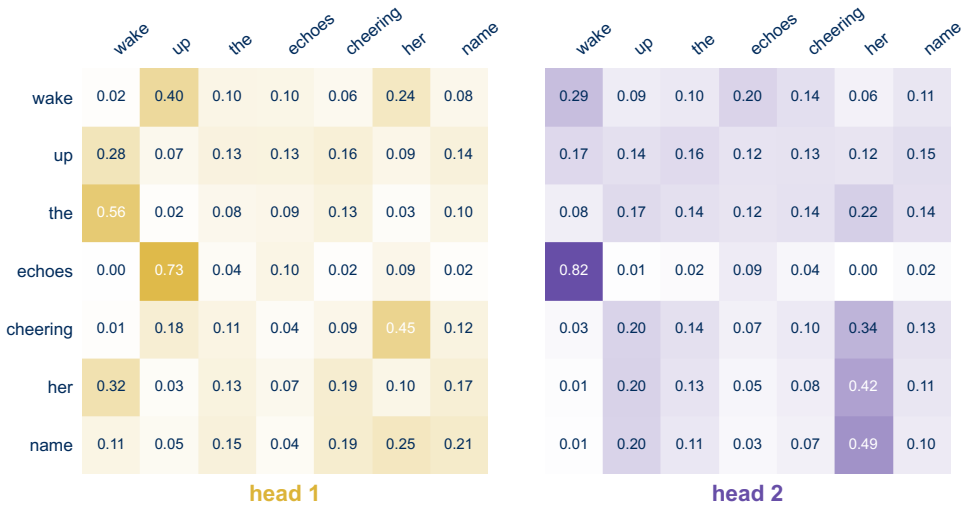
```
tensor([0.0200, 0.0900, 0.1200, 0.0200, 0.1000, 0.5100, 0.1400])
tensor([-0.2200, -0.4700, -0.2600, -0.0900])
```

Those are the weights and the output we worked out for "name".

Attention Heads

One set of W_Q, W_K and W_V gives one grid of weights, so it can only capture one kind of relevance. We can think of it as adding context for a single "thing". But a word relates to the rest of its sentence in many ways at once ("her" is the owner of "name", "echoes" is what gets woken up, "cheering" is what the echoes are doing).

So we use several sets, each called an **attention head**, side by side. Here are two heads on our line, each working with queries and keys of length `2`:



The two grids look nothing alike. In head `1` "echoes" puts `0.73` of its attention on "up", in head `2` it puts `0.82` on "wake". Between them, **"echoes" finds out it's being woken up** (at least it would, if our weights had been trained, since with random weights this is a happy accident).

Multi-Head Attention

Running several heads at once is called **multi-head attention**. It's done not with separate weight matrices, but with a logical partition of a single matrix. Our two heads use the same 4 by 4 W_Q as before, head 1 its first two columns and head 2 its last two (and likewise for W_K and W_V):

W_Q =

head 1		head 2	
0.9	1.0	0.8	0.9
-0.8	-0.3	-0.7	-0.1
0.9	0.1	0.9	0.3
-0.7	0.9	0.3	0.1

$Q_1 = XW_Q$ [:, 0:2]

$Q_2 = XW_Q$ [:, 2:4]

Each head runs the attention equation on its own slice, with $d_k = 2$, and hands back a 7 by 2 output. We concatenate the outputs side by side, back to 7 by 4, and pass them through one more linear layer, W_O, so the heads' findings get mixed together:

[5.2] h heads, each on its own slice of the columns

$$\text{MultiHead}(X) = [\text{head}_1 \ \text{head}_2 \ \cdots \ \text{head}_h]W_O$$

Splitting the columns means **more heads don't add a single weight**, they just each get a shorter slice. The model in "Attention Is All You Need" split 512 columns into 8 heads of 64, GPT-3 splits 12,288 into 96 heads of 128.

Multi-Head Attention, Continued

By hand, the heads are slices of one Q, K and V:

```
import numpy as np
import torch
rng = np.random.default_rng(28)
X = rng.uniform(-1, 1, (7, 4)).round(1) + 0.0
Ws = rng.uniform(-1, 1, (4, 4, 4)).round(1)
X = torch.tensor(X, dtype=torch.float32)
WQ, WK, WV, WO = torch.tensor(Ws, dtype=torch.float32)

def attention(Q, K, V):
    S = Q @ K.T / K.shape[1] ** 0.5
    E = torch.exp(S)
    A = E / E.sum(dim=1, keepdim=True)
    return A @ V, A

Q, K, V = X @ WQ, X @ WK, X @ WV
hs = [slice(0, 2), slice(2, 4)]  # 2 columns each
Z = [attention(Q[:, h], K[:, h], V[:, h])[0] for h in hs]
out = torch.cat(Z, dim=1) @ WO
print(out.shape, out[-1].round(decimals=2))
```

OUTPUT

```
torch.Size([7, 4]) tensor([ 0.2600, -0.3000,  0.2300,  0.2900])
```

Every token gets a new embedding the same length as the one it started with (the last is "name"'s), so the output can go straight into another round of attention.

Multi-Head Attention in PyTorch

PyTorch also has all of this as one layer, with W_Q, W_K and W_V stacked in one `12` by `4` matrix (`in_proj_weight`). Loaded with our weights, it gives the same answer:

```python
import numpy as np
import torch
from torch import nn

rng = np.random.default_rng(28)
X = rng.uniform(-1, 1, (7, 4)).round(1) + 0.0
Ws = rng.uniform(-1, 1, (4, 4, 4)).round(1)
X = torch.tensor(X, dtype=torch.float32)
WQ, WK, WV, WO = torch.tensor(Ws, dtype=torch.float32)

mha = nn.MultiheadAttention(embed_dim=4, num_heads=2,
                            batch_first=True)
W = torch.cat([WQ.T, WK.T, WV.T])  # 12 by 4
with torch.no_grad():  # our weights, no biases
    mha.in_proj_weight.copy_(W)
    mha.out_proj.weight.copy_(WO.T)
    mha.in_proj_bias.zero_()
    mha.out_proj.bias.zero_()
Z, A = mha(X[None], X[None], X[None])  # a batch of 1
print(Z.shape, Z[0, -1].detach().round(decimals=2))
```

OUTPUT

```
torch.Size([1, 7, 4]) tensor([ 0.2600, -0.3000,  0.2300,  0.2900])
```

PyTorch multiplies by the transpose of each weight matrix, so we load W_Q^T and the rest.

Word Order

A **Generative Pre-trained Transformer** (GPT) is, for the most part, multi-head self-attention stacked many times over, each layer's output going in as the next layer's input (GPT-3 stacks `96`). Nowhere in it is there an RNN, and nothing has to wait for the token before.

Unfortunately, that has a cost. Let's shuffle our line into "the cheering echoes name her wake up" and run the same attention on it. Every token gets exactly the same output it got before, just in a different row. "name" still gives `0.51` to "her", although "her" now comes after it.

Nothing in the equation looks at where a token is, only at what it is. Self-attention treats a sentence as a bag of words, so **"wake up the echoes" and "the echoes wake up" look identical to it**. The RNN got word order for free, by reading in order, and that's exactly the part we threw away.