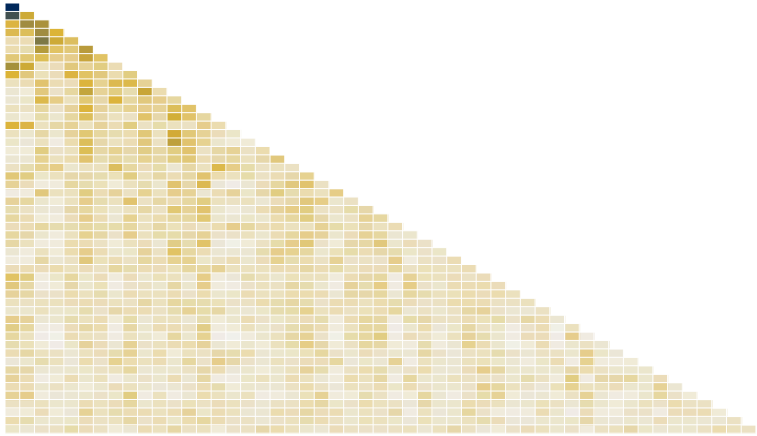
UNIT 03 · CHAPTER 6

# Transformers

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



# Generative Pre-trained Transformer (GPT)

**ChatGPT** writes its answer **one word at a time**: each word is a guess at what comes next, given everything before it

**cheer** **cheer** **for** **old** **notre** _____

Guessing the next word from the words before it is called _____, and it's all a **GPT** does: a **Generative** (it writes) **Pre-trained** (on lots of text) **Transformer**

Monday's **self-attention** mixes every word's **context** into its **embedding**, but it **doesn't write anything**. Today: from that layer to a **GPT** that writes the **fight song**

# Next-Token Prediction

The first line of the chorus is already `5` guesses to learn from:

| the words so far | the next word |
|---|---|
| cheer | cheer |
| cheer cheer | for |
| cheer cheer for | old |
| cheer cheer for old | notre |
| cheer cheer for old notre | dame |

The whole chorus is `50` training examples, one for every word after the first, and **nobody had to label anything**: the answer is just the **next word**

Using raw text like this, without needing to label ground-truth data, is called

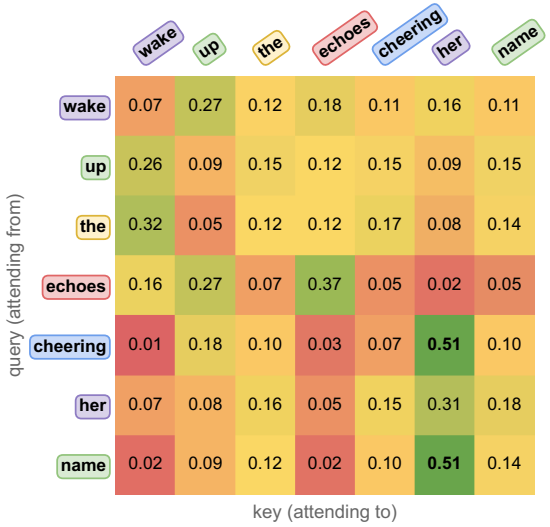
_____

# Self-Attention

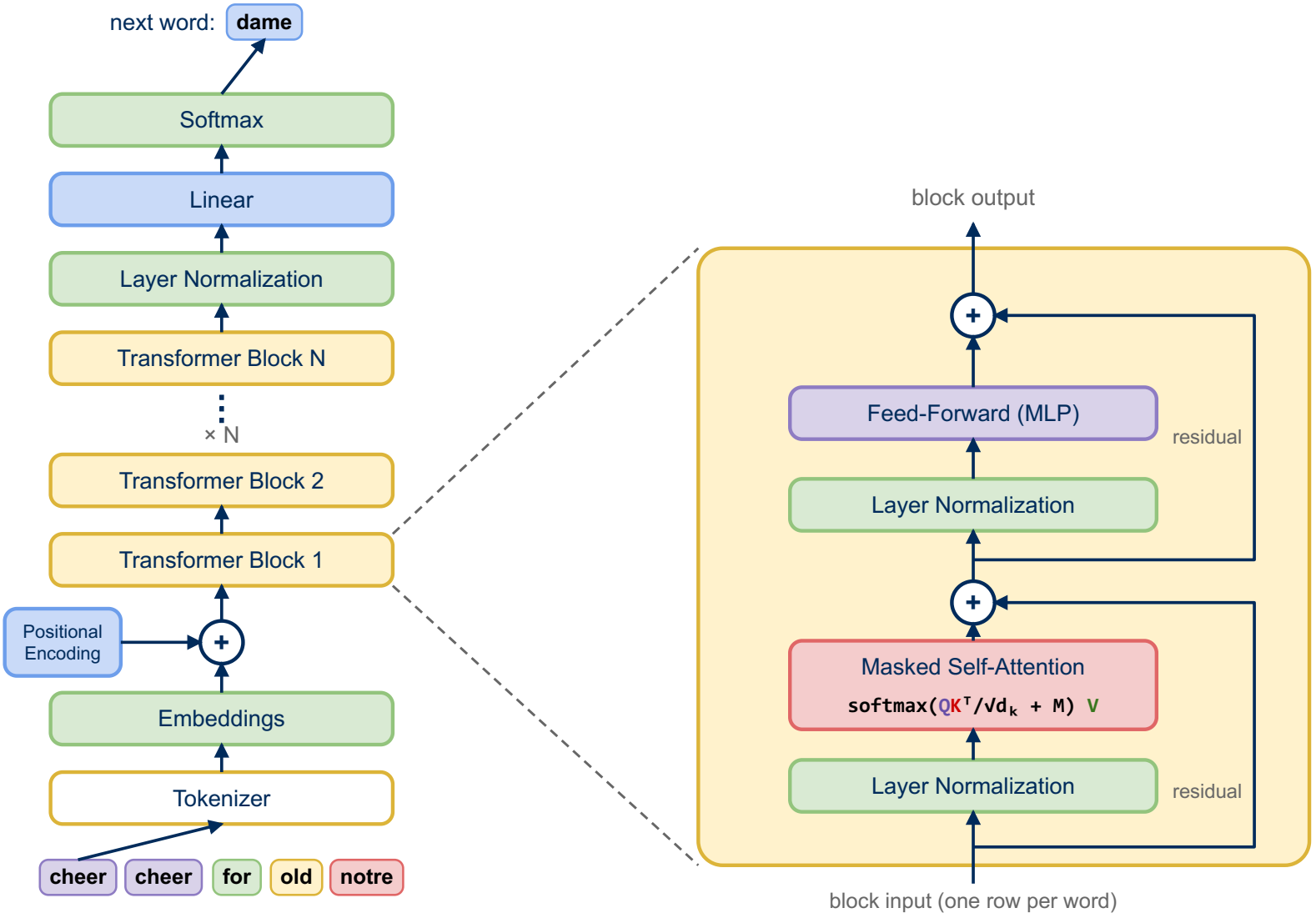
[6.1] scaled dot-product attention

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

- $Q = X\,W_Q$, the _____: what this **token** is **looking for**
- $K = X\,W_K$, the _____: what it **offers to be matched against**
- $V = X\,W_V$, the _____: what it **passes along**



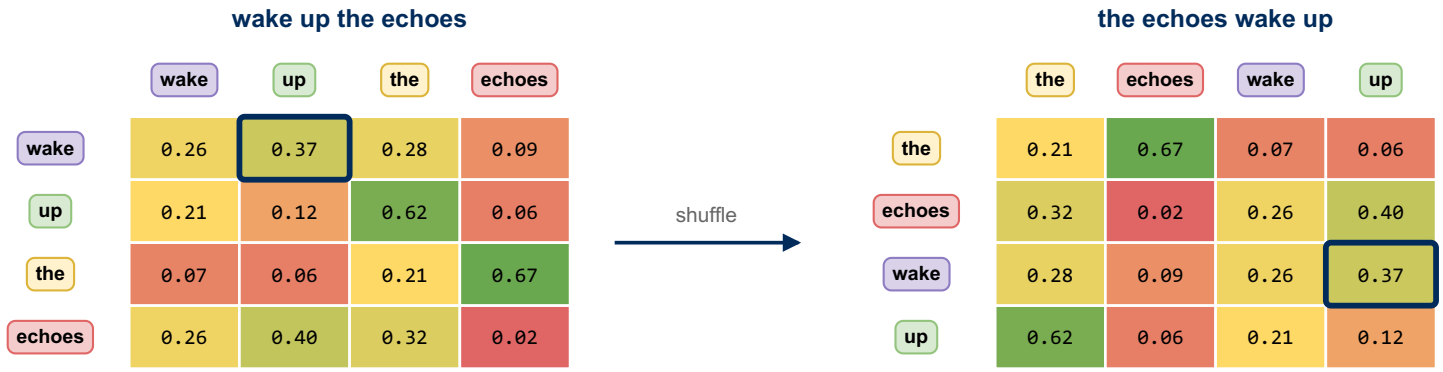
# The Whole Model



# Self-Attention and Word Order

Shuffle our line: **the** **cheering** **echoes** **name** **her** **wake** **up**. Every **token** gets **the same output, just in a different row**

**Self-attention** treats a sentence as a **bag of words**: **wake up the echoes** = **the echoes wake up**



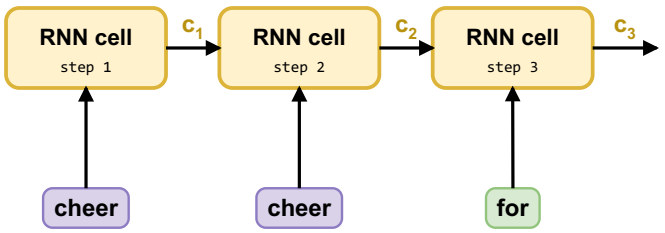

**wake → up is 0.37 in both: every weight is the same, just moved to another row and column**

one row per query, one column per key (no positions)

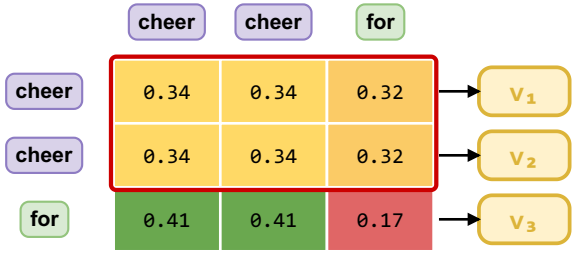
# Positional Embeddings

Same problem in the chorus: it starts **cheer** **cheer**, and both get exactly the same output. **Attention can't tell the first cheer from the second**

**RNN: one word at a time**




**Attention: every word at once**




the two cheer rows are the same, so $v_1 = v_2$

one row per query, one column per key

An **RNN** got the order for free (it read the first **cheer** first). **Attention** is computed _____, so each word's position is lost

# Positional Embeddings

So we put the position back in ourselves: each **token**'s position goes through _____ functions, each column with a different frequency, and the result is **added to its embedding**

[6.2] even columns

$$P_{pos,\,2i} = \sin\left(pos \,/\, 10000^{2i/d}\right)$$

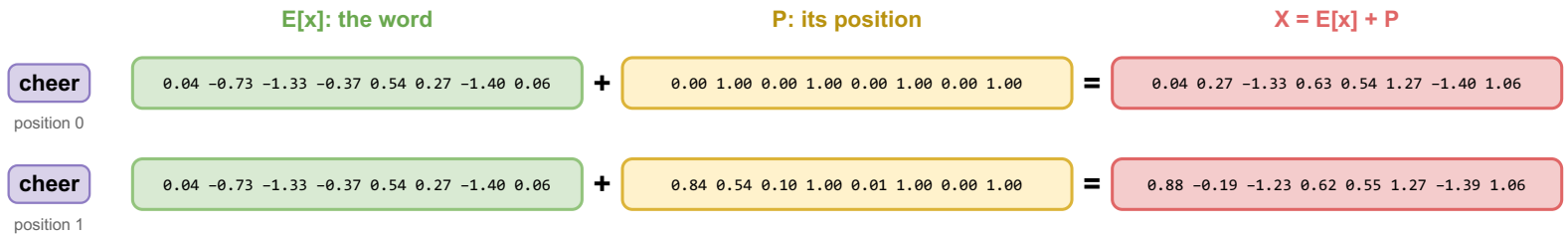
odd columns

$$P_{pos,\,2i+1} = \cos\left(pos \,/\, 10000^{2i/d}\right)$$

[6.3] what the first block sees

$$X = E_x + P$$

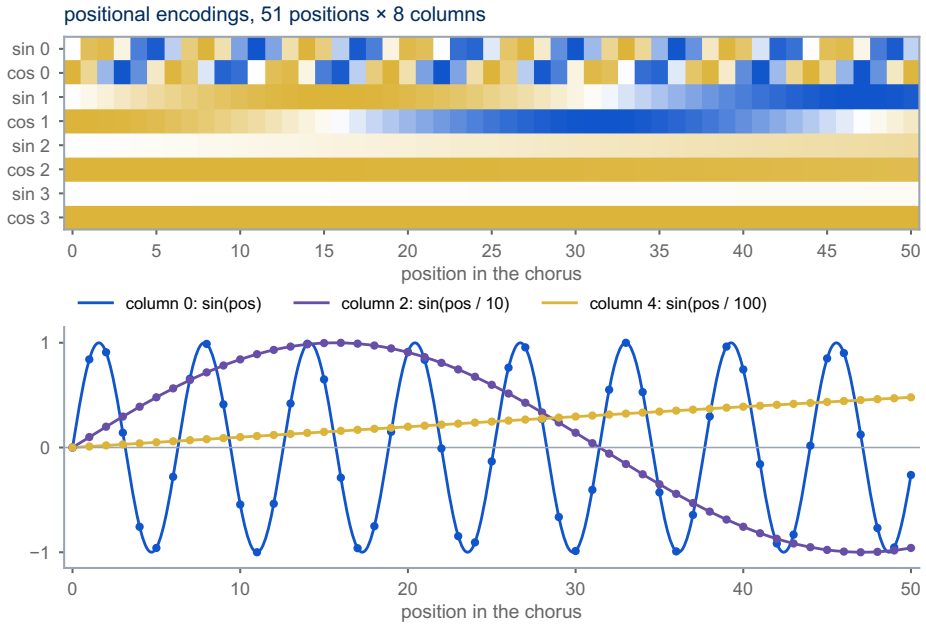
This is referred to as **Sinusoidal Position Encoding**




**NOTE:** Modern LLMs use something called RoPE that doesn't actually modify the original embeddings

# Positional Embeddings

Every position gets **its own pattern**: the first columns flip every word (telling **neighbors** apart), the last ones barely move (telling **the start of the chorus from the end**), like the second, minute and hour hands of a clock




So the two **cheer**s now go in as **different positions**

# Training on Every Word at Once

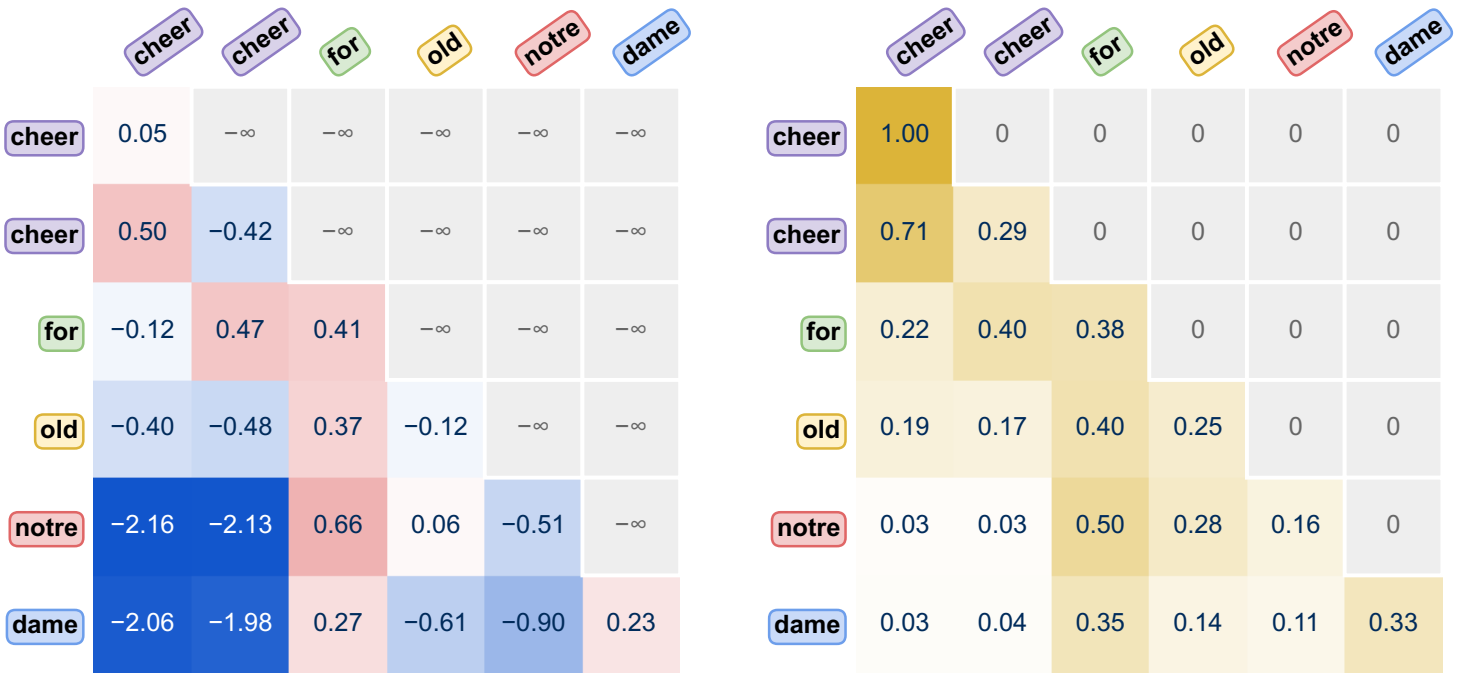
We don't want to run the model `50` times. Instead we give it the whole chorus and ask **every row to guess the word after it**, `50` guesses in one pass:

| | | |
|---|---|---|
| **in** | **cheer** **cheer** **for** **old** **notre** | … |
| **guess** | **cheer** **for** **old** **notre** **dame** | … |

Unfortunately, **attention** lets every word look at every other word, so the row for **notre** could just look at **dame** and copy it: **the model would learn to cheat**

# Masking

Before the **softmax** in **self-attention**, every score where a **token** looks at a **later token** is set to **−∞**



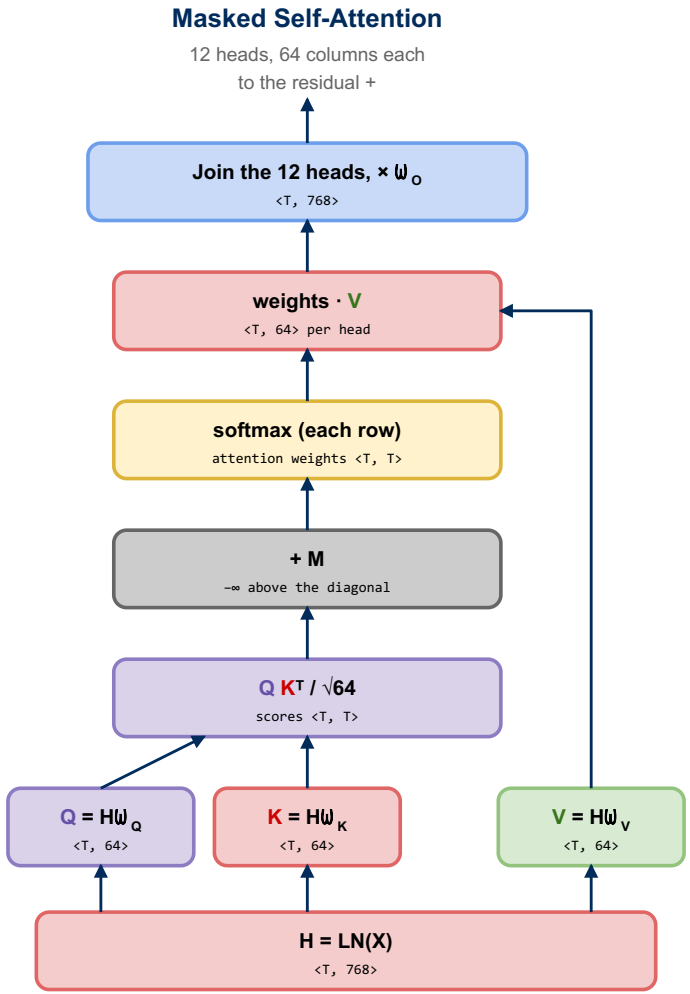

scores $Q K^T / \sqrt{8}$, masked

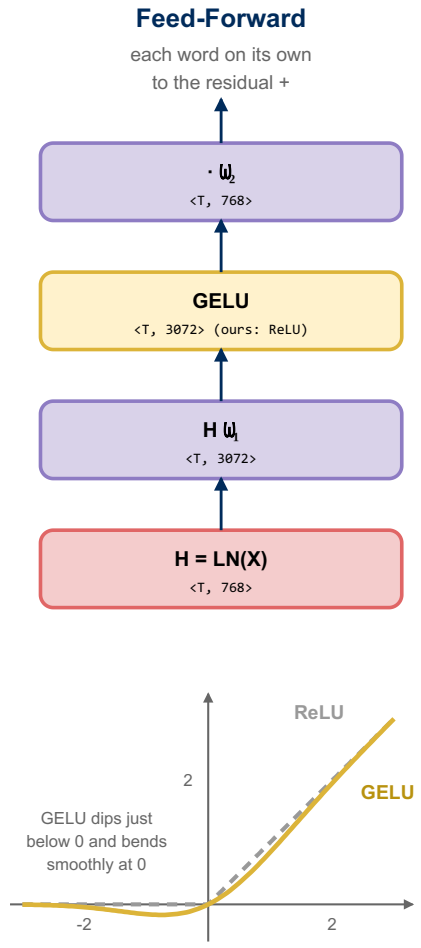
attention weights (softmax of each row)

**NOTE:** $e^{-\infty} = 0$, so **notre** weighs the words up to itself (**0.03**, **0.03**, **0.50**, **0.28**, **0.16**), and **dame** gets **0**

# Simple Decoder/Generative Transformer Diagram

The **mask** lives in the block's **attention**. One **transformer block**, at **GPT-2** small's size:

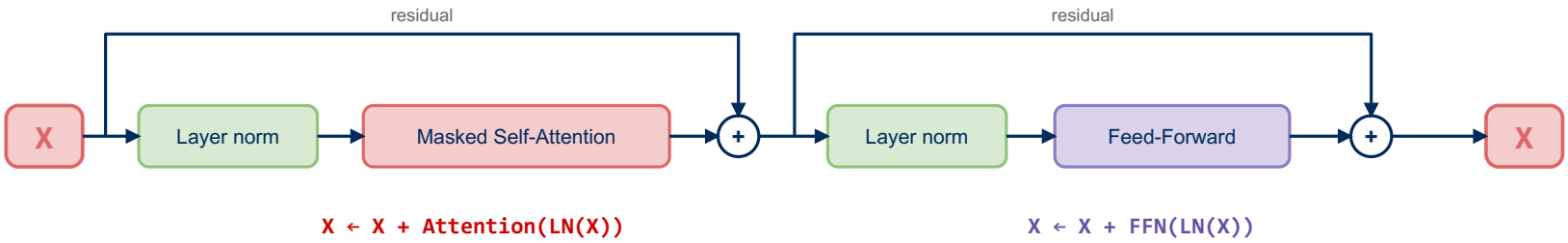


# Residual Connections, Again

Monday we added Z back onto X. A **transformer block** does that twice, once around each half:

$$X \leftarrow X + \text{Attention}(\text{LN}(X))$$
$$X \leftarrow X + \text{FFN}(\text{LN}(X))$$




Each half only **adds a change** to every word's row, the word itself is still there: a

---

A block's output has **exactly the same shape as its input**, a row of d numbers per word

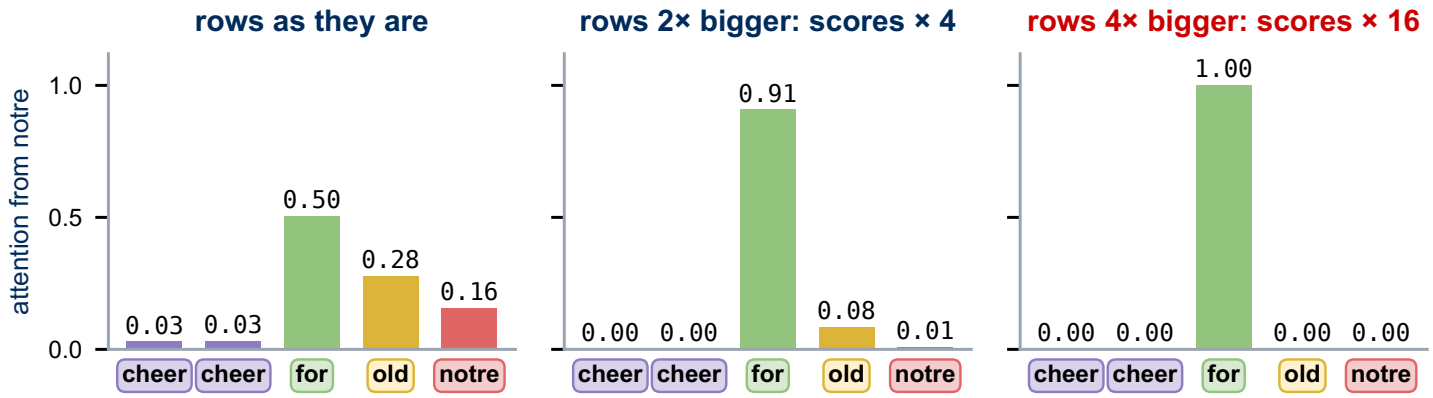
# Layer Normalization

Before each half, every word's row is rescaled to **mean 0 and standard deviation 1**:

$$\mathrm{LN}(x) = \frac{x - \mathrm{mean}(x)}{\mathrm{std}(x)}$$

| dame | $x_1$ | $x_2$ | $x_3$ | $x_4$ | $x_5$ | $x_6$ | $x_7$ | $x_8$ | mean | std |
|---|---|---|---|---|---|---|---|---|---|---|
| before | -1.86 | -0.17 | -0.09 | 0.42 | -0.47 | 1.73 | 0.48 | -0.56 | −0.06 | 0.96 |
| after | -1.87 | -0.11 | -0.02 | 0.50 | -0.42 | 1.86 | 0.57 | -0.52 | 0 | 1 |

Why: every block **adds onto the rows**, so their scale drifts as the blocks stack up, and big numbers make **softmax** give **nearly all attention to one token**



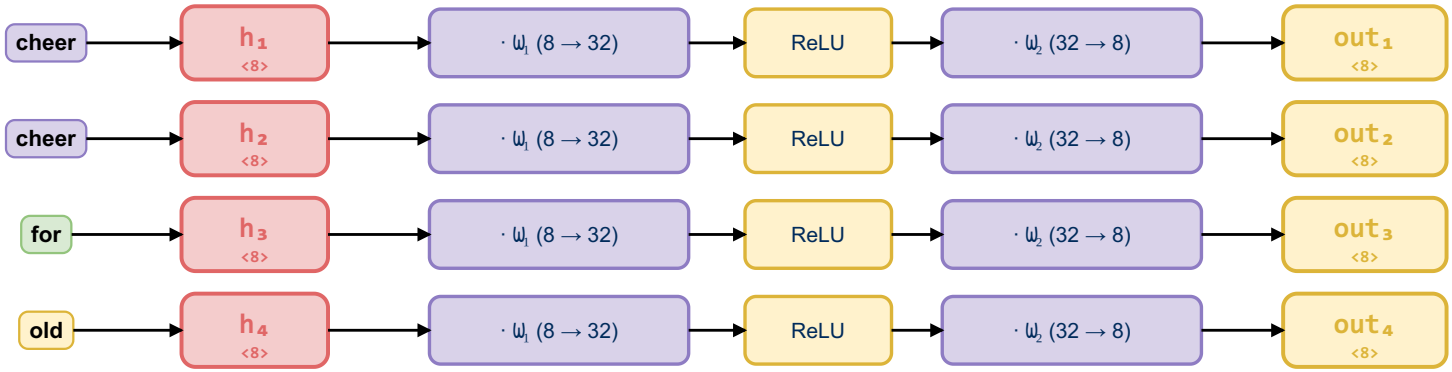
# The Feed-Forward Sublayer

**Attention** mixes the words together. Then a small **feed-forward network** works on **each word by itself**:

$$\mathrm{FFN}(H) = \mathrm{ReLU}(HW_1)\,W_2$$

- **Ours**: `8` → `32` → `8` numbers per word
- **GPT-2**: `768` → `3,072` → `768`, `4.7` million of each block's `7.1` million weights

It's the **feed-forward network** from a few weeks ago, run on every word's row on its own

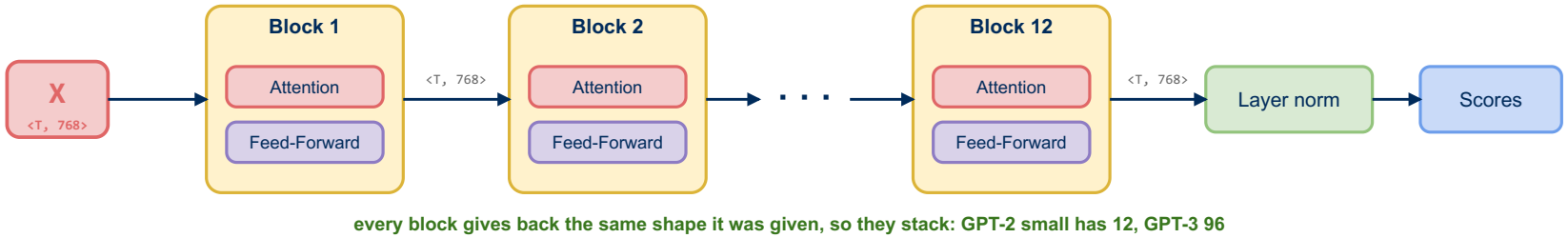



the same $W_1$ and $W_2$ for every word, and no arrows between the rows: each word on its own

# Stacking Blocks

A block gives back rows of **the same shape it was given**, so its output can go straight into another block, and another after that:

- **Ours**: 1 block
- **GPT-2** small: 12 blocks, the largest **GPT-2**: 48
- **GPT-3**: 96 blocks




every block gives back the same shape it was given, so they stack: GPT-2 small has 12, GPT-3 96

Each block mixes the words again, so later blocks work on rows that **already know about their neighbors**

# Our Tiny GPT in PyTorch

Our **tiny GPT**: the chorus as **41** different words, **8** numbers per word, and the **positional encoding** from earlier, in the same colours

```python
import torch
from ai2ai import load
song = load("fight-song")
print(song)

def pos_enc(T, d=8):
    pos = torch.arange(T)[:, None]
    i = torch.arange(0, d, 2)
    angle = pos / 10000 ** (i / d)
    P = torch.zeros(T, d)
    P[:, 0::2] = torch.sin(angle)
    P[:, 1::2] = torch.cos(angle)
    return P
```

OUTPUT

```
51 words, 41 different ones
```

$$P_{pos,\,2i} = \sin\left(pos \,/\, 10000^{2i/d}\right)$$
$$P_{pos,\,2i+1} = \cos\left(pos \,/\, 10000^{2i/d}\right)$$

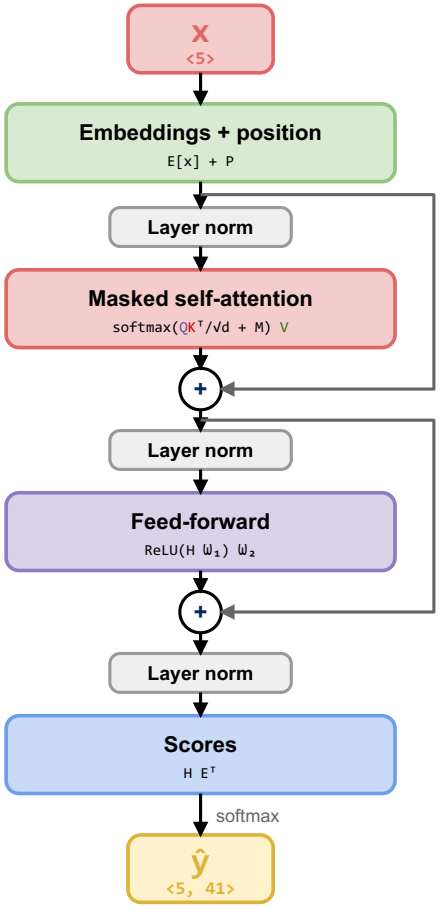
# Layer Norm and the Weights

One **transformer block**'s weights, loaded **before training**, coloured by the part of the model that uses them, plus the **layer norm** (grey)

```
1  W = load("tiny-gpt", trained=False)
2
3  def norm(X):
4      mean = X.mean(1, keepdim=True)
5      var = ((X - mean) ** 2).mean(1, keepdim=True)
6      return (X - mean) / var.sqrt()
```

OUTPUT

```
E   (41, 8)
Q   (8, 8)
K   (8, 8)
V   (8, 8)
O   (8, 8)
F1  (8, 32)
F2  (32, 8)
```



# The Output Layer

```python
def gpt(x, W, mask=True):
    T = len(x)
    X = W.E[x] + pos_enc(T)

    H = norm(X)

    Q, K, V = H @ W.Q, H @ W.K, H @ W.V

    S = Q @ K.T / 8 ** 0.5

    if mask:
        S = S + torch.full((T, T), float("-inf")).triu(1)

    attention_scores = torch.softmax(S, 1)
    context_vectors = attention_scores @ V
    X = X + context_vectors @ W.O

    X = X + torch.relu(norm(X) @ W.F1) @ W.F2

    return norm(X) @ W.E.T

x = song.ids[:5]        # cheer cheer for old notre

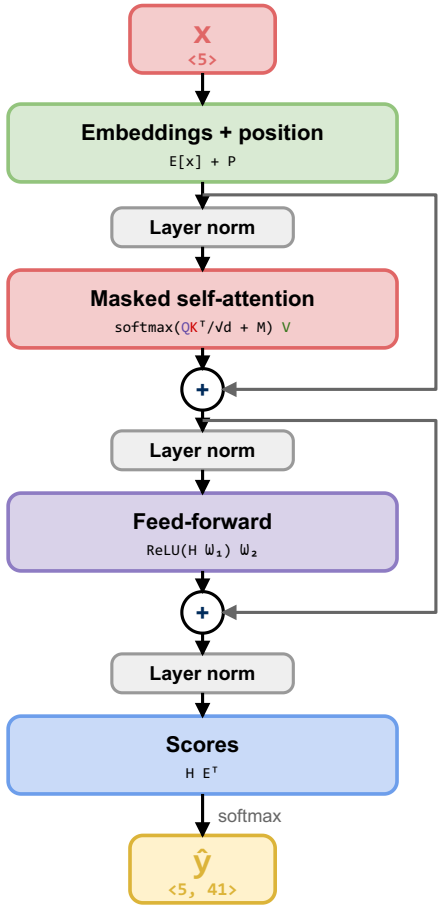
probs = torch.softmax(gpt(x, W)[-1], 0)

guess = probs.argmax().item()
dame = song.vocab.index("dame")
p = round(probs[guess].item(), 2)

print(song.decode([guess]), p)
print(song.decode([dame]), round(probs[dame].item(), 4))
```

OUTPUT

```
[odds] 0.26
[dame] 0.0008
```




Untrained: 26% on **odds**, 0.08% on **dame**, **a thirtieth** of guessing evenly (2.4%)

# Training Our Tiny GPT

The loss is _____, but with 41 classes (**tokens** in our vocabulary)

```python
import torch.nn.functional as F
opt = torch.optim.Adam(W.params(), lr=0.03)
x, y = song.ids[:-1], song.ids[1:]    # y: the next word

for step in range(301):
    loss = F.cross_entropy(gpt(x, W), y)

    opt.zero_grad()
    loss.backward()
    opt.step()

    if step % 100 == 0:
        print(f"step {step}  loss {loss.item():.3f}")
```

OUTPUT

```
step 0  loss 7.513
step 100  loss 0.077
step 200  loss 0.024
step 300  loss 0.013
```

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

$$W \leftarrow \text{Adam}\left(W, \frac{\partial \mathcal{L}}{\partial W}\right)$$

From **7.513** to **0.013** in 300 steps of **Adam**

# Generating Text

Pick the most likely word, add it to the end, and repeat:

This is called _____ with _____

```python
def generate(W, start, n, mask=True):
    x = song.encode(start)
    for _ in range(n):  # greedy decoding
        scores = gpt(torch.tensor(x), W, mask)

        x.append(scores[-1].argmax().item())

    return song.decode(x)

print(generate(W, "cheer cheer for old notre", 46))
```

OUTPUT

```
[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]
```

$$\text{scores} = \text{GPT}(x_1, \ldots, x_T)$$

$$x_{T+1} = \arg\max \text{scores}_T$$

**Our GPT writes the fight song**. It memorized 51 words (**GPT-3** learned from about 300 billion **tokens**), but it's the same loop that writes every **ChatGPT** answer

# Without the Mask

Same model, same training, **no mask**: the red part is gone

```
Wn = load("tiny-gpt-nomask")   # trained without it

start = "cheer cheer for old notre"

print(generate(Wn, start, 46, mask=False))
```

OUTPUT

```
[cheer, cheer, for, old, notre, dame, shake, a,
volley, up, the, wake, up, the, wake, up, the,
wake, up, dame, wake, up, volley, on, be, great,
a, volley, cheer, while, notre, dame, will, win,
win, win, win, win, win, win, win, win, notre,
dame, will, win, win, notre, dame, and, dame]
```

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}} + M\right)V$$

The loss gets just as low, because during training every row could **peek at the answer**.

When it writes, **there's no next word to peek at**, so it falls apart after **dame**