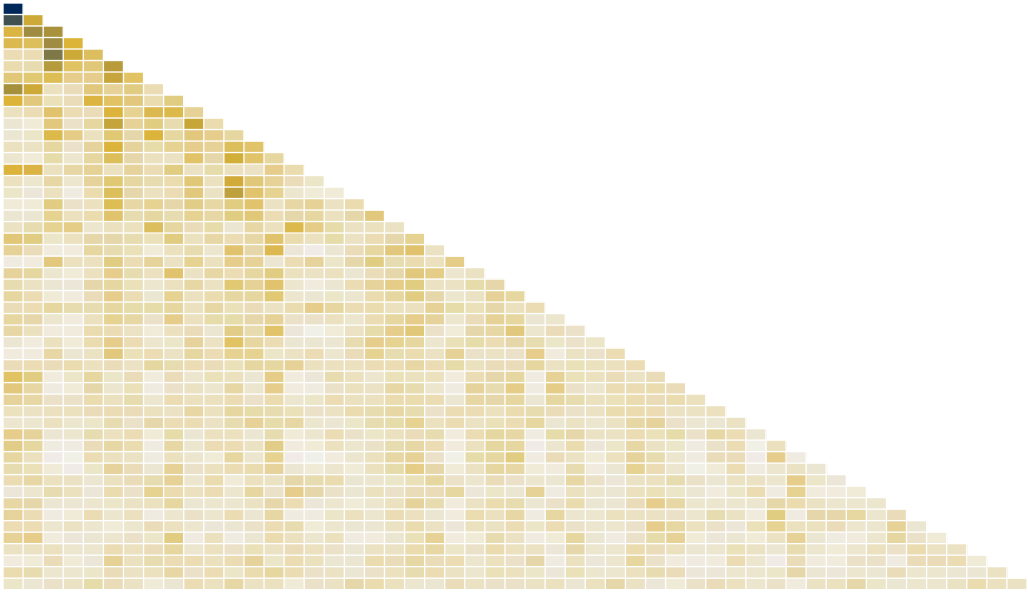


UNIT 03 · CHAPTER 6

Transformers

CSE 30124 · Introduction to Artificial Intelligence

William Theisen · University of Notre Dame

Generative Pre-trained Transformers

In the last chapter we built **self-attention**, where every word gets back a weighted average of the other words' values, so the "dame" in "Notre Dame" can end up knowing about the "notre" in front of it.

What attention hands back is just another row of numbers for every word. It doesn't write anything. The models that do write are built by stacking attention into a **transformer**, the best known being OpenAI's GPT, short for Generative Pre-trained Transformer (the first came out in `2018`).

Our example is the chorus of the Victory March, written in `1908` by Michael and John Shea:

> *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.*

Anyone at a game can finish "Cheer, cheer for old Notre …". So given the words so far, our goal is to get a computer to do the same.

Next-Token Prediction

Guessing the next word from the words before it is called **next-token prediction** (a token is usually a piece of a word, but in our example every token will be a whole word). The first line of the chorus already gives us five guesses to make:

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

Notice that a piece of text is a whole set of training examples. Our `51`-word chorus gives us `50`, one for every word after the first, and **nobody had to label anything**, since the answer is just the next word of the text.

That's the "pre-trained" part of the name. A GPT is first trained on this one task over an enormous amount of text (GPT-2's training set, WebText, was about `40` GB from `8` million web pages), before it's pointed at anything in particular. Generating text is then the same guess over and over, with each new word added to the end of the input.

Tokens and Embeddings

As in the last chapter, every word is a token, and this time each different word also gets an id:

```python
import re

chorus = """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 = re.findall(r"[a-z]+", chorus.lower())
vocab = list(dict.fromkeys(words))  # no repeats
ids = [vocab.index(w) for w in words]
print(len(words), len(vocab))
print(words[:6])
print(ids[:6])
```

OUTPUT

```
51 41
['cheer', 'cheer', 'for', 'old', 'notre', 'dame']
[0, 0, 1, 2, 3, 4]
```

The chorus has `51` tokens but only `41` different words, each with an id that picks one row of an **embedding matrix** E (the last chapter's X was seven of these rows). Our model for the rest of the chapter is tiny, with d = 8 numbers per word, so E is `41 × 8`. GPT-2 small has `50,257` tokens (many of them word pieces like "ing") and d = 768, so its E alone is `38,597,376` numbers.

Word Order

As we saw at the end of the last chapter, self-attention pays for working on every word at once by ignoring word order. Our tiny model is no different. Take the line "old notre dame will win over all", run it through our model's attention, and look at the output for "dame":

in order
```
[0.42 −0.26 0.80 0.01 0.14 −0.06 −0.25 −0.70]
```

Now shuffle the line into "all win dame old will over notre" and run it again. The output for "dame" comes back exactly the same, to `15` decimal places.

Repeated words are a problem too. Our chorus starts "cheer cheer", and both cheers get exactly the same output, so **attention can't tell the first cheer from the second**. A recurrent network got this for free, since it read the first one first.

Positional Encodings

So we have to put the position in ourselves. The original transformer paper (Vaswani et al., **2017**) passes each token's position through sine and cosine functions, each column with a different frequency, and **adds the result to the token's embedding**:

[6.1] even columns

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

odd columns

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

[6.2] what the first block sees

$$X = E_x + P$$

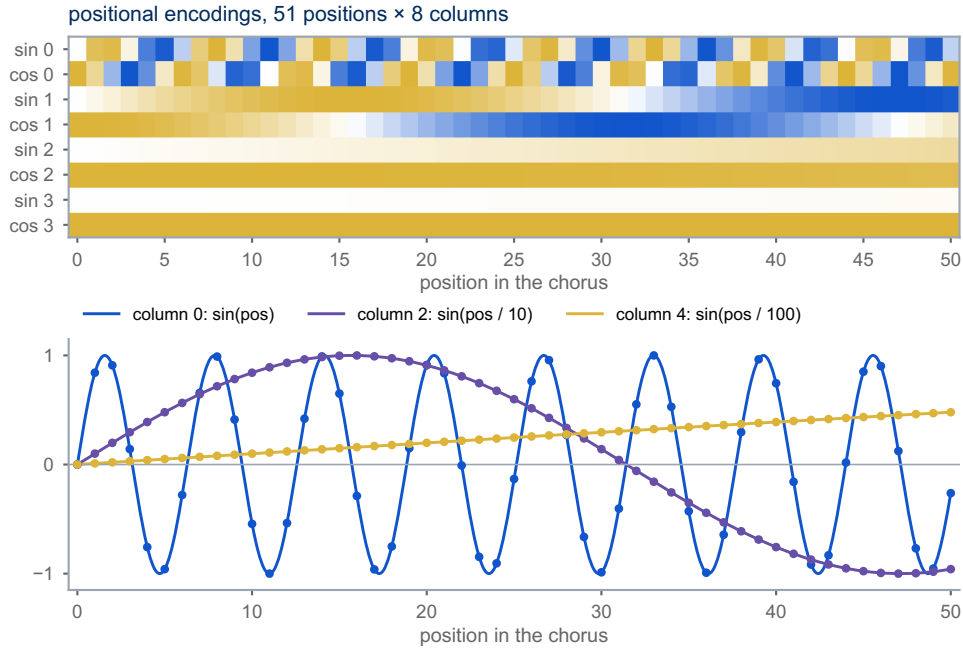
```python
import torch
def pos_enc(T, d):
    pos = torch.arange(T)[:, None]    # 0, 1, ..., T-1
    ang = pos / 10000 ** (torch.arange(0, d, 2) / d)
    P = torch.zeros(T, d)
    P[:, 0::2] = torch.sin(ang)       # even columns
    P[:, 1::2] = torch.cos(ang)       # odd columns
    return P
for row in pos_enc(6, 8):
    print([round(v, 2) for v in row.tolist()])
```

OUTPUT

```
[0.0, 1.0, 0.0, 1.0, 0.0, 1.0, 0.0, 1.0]
[0.84, 0.54, 0.1, 1.0, 0.01, 1.0, 0.0, 1.0]
[0.91, -0.42, 0.2, 0.98, 0.02, 1.0, 0.0, 1.0]
[0.14, -0.99, 0.3, 0.96, 0.03, 1.0, 0.0, 1.0]
[-0.76, -0.65, 0.39, 0.92, 0.04, 1.0, 0.0, 1.0]
[-0.96, 0.28, 0.48, 0.88, 0.05, 1.0, 0.0, 1.0]
```

Why Sines and Cosines

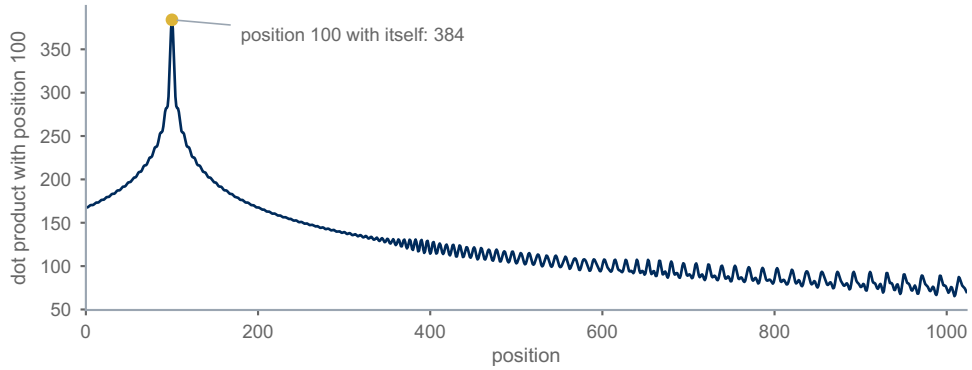
Row `0` is added to the first "cheer" and row `1` to the second, so the two cheers now go into attention as different rows (starting `0.04`, `0.27`, … and `0.88`, `-0.19`, …) and come out different.



Pictured above are the encodings for all `51` positions of the chorus. The first two columns repeat about every `6` positions, so on their own they'd mix up position `0` with position `44`. The slower columns have barely started their swing by then, so **no two positions get the same row**, a bit like the hands of a clock.

Relative Position

The sines and cosines have one more useful property, which shows up at GPT-2's size. Pictured below is the dot product of position `100`'s encoding with every other position's, for d = 768.



It's `384` with itself, `373.8` with its neighbor and `167.8` with position `200`. Positions `500` and `501` also give `373.8`, so **how alike two positions look depends only on how far apart they are**, which is usually what matters in a sentence ("dame" comes right after "notre" wherever the line falls in the song).

Why this recipe rather than some other is a fair question, and the honest answer from the slides is "somehow this works". GPT-2 doesn't use it at all. It learns a **position embedding** for each of its `1,024` positions instead (another `786,432` weights), and adds those.

Causal Masking

Training on next-token prediction has a catch. We don't want to run the model 50 times, once for each start of the chorus, so we give it every word at once and ask every position to guess the word after it. Unfortunately, attention lets every word look at every other word, including the one it's meant to be guessing. The row for "notre" could just look at "dame" and copy it, and **the model would learn to cheat**.

The fix is called **masking**. Before the softmax in the last chapter's attention equation, every score where a word looks at a later word is set to $-\infty$:

a word may look at itself and earlier words

[6.3]
$$M_{ij} = \begin{cases} 0 & j \le i \\ -\infty & j > i \end{cases}$$

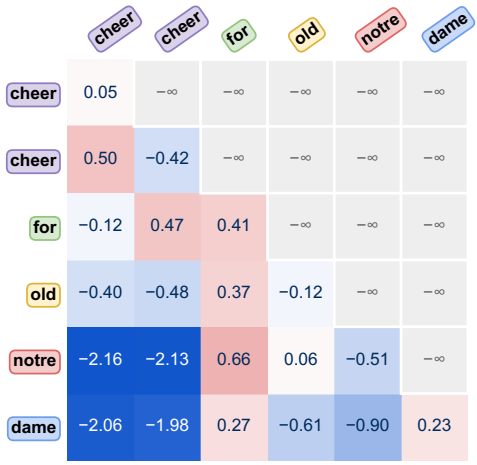
masked self-attention

[6.4]
$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}} + M\right)V$$

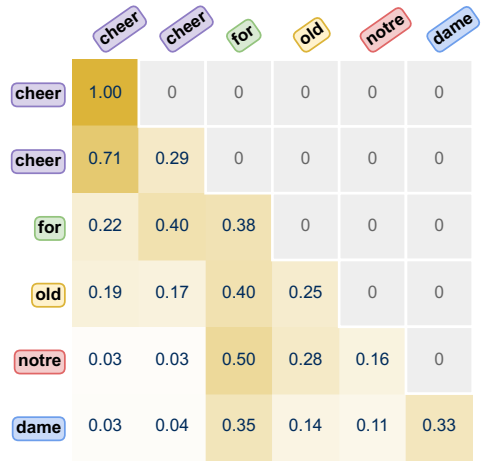
Since $e^{-\infty} = 0$, every masked weight comes out exactly 0 after the softmax, and each row still sums to 1 over the words it's allowed to see. (Our model has one head, so its d_k is all 8 columns.)

Masked Attention Weights

Here is the mask at work on the first six words of the chorus in our tiny model.



scores Q K^T / √8, masked



attention weights (softmax of each row)

Each row is a word doing the looking and each column a word being looked at. On the right, "notre" spreads its attention over the five words up to itself (`0.03`, `0.03`, `0.50`, `0.28`, `0.16`) and gives "dame" nothing, and the first "cheer" can only look at itself, so it puts all of its weight there.

Only the lower triangle survives, and every word's output depends only on itself and the words before it, so this is called a **causal mask**. Our weights are random, so the `0.50` on "for" doesn't mean anything yet. The mask is the same whatever the weights are, so training can't learn its way around it.

Residual Connections

Attention gives back, for every word, a weighted average of the values of the words it can see. If that became the word's new row, "notre" would be replaced by something about half "for" (its biggest weight, **0.50**) and only **16%** itself. So instead of replacing each row, we add the attention output on top of it:

[6.5] add, don't replace

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

This is called a **residual connection** (or skip connection). The attention only has to work out what to change, and everything the row already held, the word and its position, carries on through. For "notre" the row going in has a length of **2.62**, and attention adds a vector of length **1.31** to it.

Residual connections came from image recognition (He et al.'s ResNet, **2015**), where they made it possible to train networks over a hundred layers deep. The reason is backpropagation. The derivative of $x + f(x)$ with respect to x is $1 + f'(x)$, so even when f's gradient is tiny, the gradient flowing back through the + doesn't vanish.

Layer Normalization

Adding keeps everything, but the rows keep growing. Our average row has a length of `2.84` going in and `3.36` once attention is added. GPT-2 small adds onto its rows `24` times, each time changing their scale.

Layer normalization fixes the scale of each row before a sublayer (such as attention) reads it, by subtracting the row's mean and dividing by its standard deviation:

[6.6]

one row x, with mean μ and standard deviation σ

$$\mathrm{LN}(x) = \gamma \frac{x - \mu}{\sigma} + \beta$$

γ and β are learned, one of each per column, so a trained model can undo some of it where that helps (ours are `1` and `0`). The "notre" row once attention has been added, for example:

mean 0.55, standard deviation 1.08

```
[-1.15  0.65  1.58  0.33  2.68  0.56 -0.07 -0.15]
  ↓ LN
[-1.57  0.09  0.95 -0.21  1.96  0.01 -0.57 -0.65]
```

Only the sublayer gets the normalized copy, the rows are left alone. GPT-2 normalizes at the input of each sublayer like this (the original transformer did it after each add), so the rows behind our mask figure had already been through one.

The Feed-Forward Sublayer

Attention is the only place the words share anything, and it's mostly averaging (weights from a softmax, then a weighted sum of values). The other sublayer works on each word on its own. It's a small **feed-forward network**, the same kind we started this unit with, with one hidden layer:

[6.7] the same weights for every row

$$\mathrm{FFN}(H) = \max(0,\ HW_1)\,W_2$$

The same W_1 and W_2 are applied to every row separately, so "notre" never sees "dame" here (or any other word). Ours goes from 8 columns up to 32 and back down to 8, and the ReLU zeroes out between 13 and 21 of the 32 hidden units, depending on the word.

GPT-2 small goes from 768 up to $3{,}072$ (four times wider, the usual choice) and back, with biases and GELU, a smoothed ReLU. That's $4{,}722{,}432$ weights per block, **about two thirds of the whole block**. A lot of what a trained model "knows" about words seems to be stored in these weights.

The Transformer Block

Put together, a **transformer block** is a masked attention sublayer and a feed-forward sublayer, each reading a layer-normed copy of the rows:

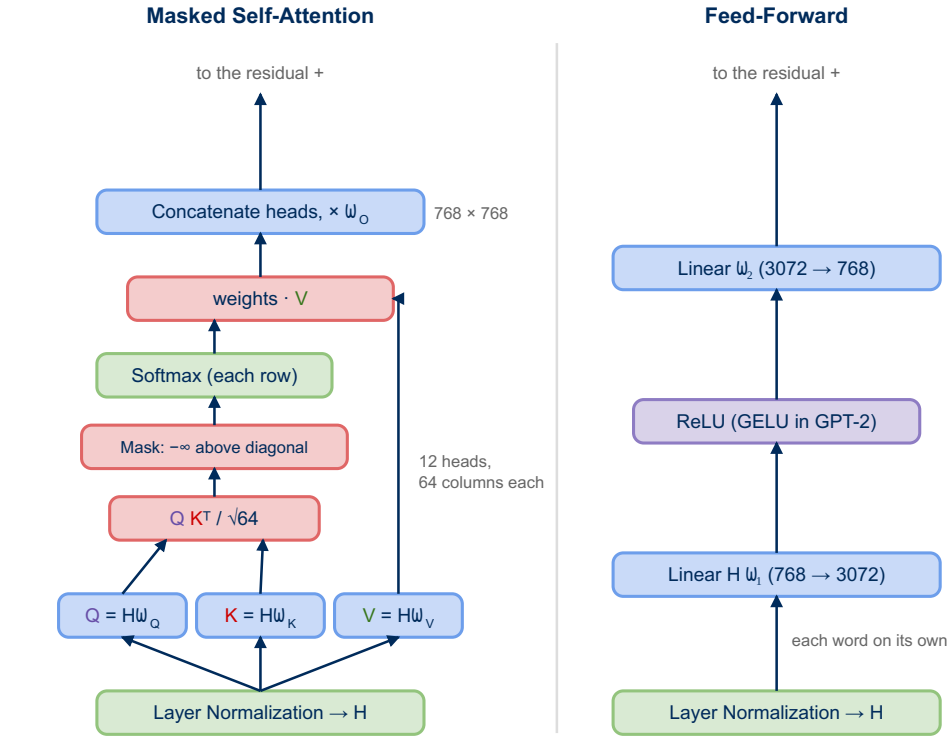
[6.8]

attention sublayer

$$X \leftarrow X + \texttt{Attention}(\texttt{LN}(X))$$

feed-forward sublayer

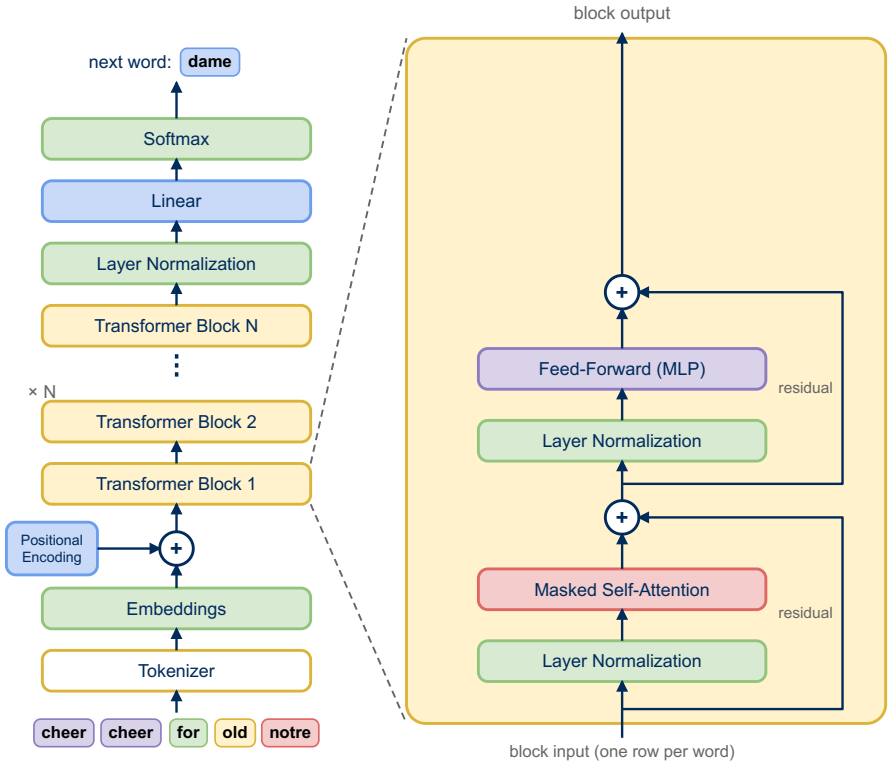
$$X \leftarrow X + \texttt{FFN}(\texttt{LN}(X))$$



Pictured above are the two sublayers at GPT-2 small's size, with **12** attention heads of **64** columns each. A block's output has **exactly the same shape as its input**, a row of d numbers per word.

Stacking Blocks

Since a block gives back rows of the same shape it was given, the output of one block can go straight into another, and another after that. Pictured below is the whole model.



Each block gets to mix the words again, so later blocks work on rows that already know about their neighbors. Ours has one block, GPT-2 small has `12`, and the largest GPT-2 has `48`. Because there's no encoder in front of it, this is called a **decoder-only transformer** (the original `2017` transformer, built for translation, had both).

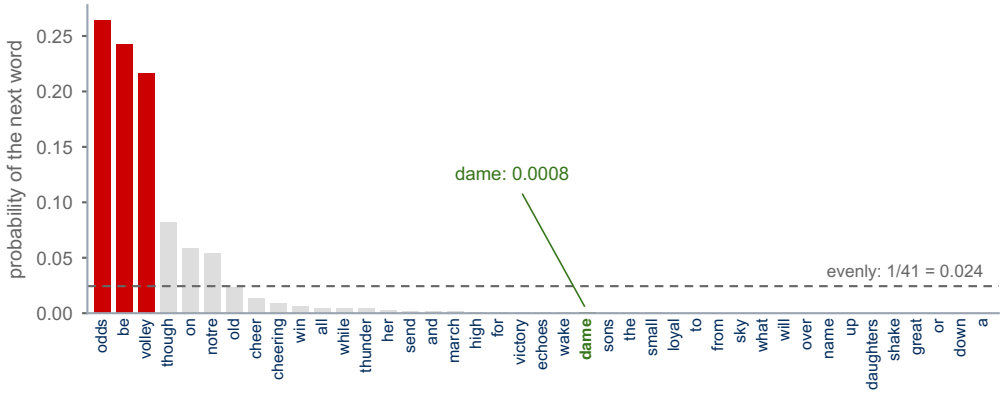
The Output Softmax

To turn the rows into guesses, they get one more layer norm and are multiplied by the transposed embedding matrix, which gives every position one score per word in the vocabulary, and a softmax of each row turns the scores into probabilities:

[6.9] a row per position, a probability per word in the vocabulary

$$\hat{y} = \mathrm{softmax}(\mathrm{LN}(X)\,E^{T})$$

Reusing **E** like this is called **weight tying** (GPT-2 does it), and it means each word's score is the dot product of a row with that word's own embedding.



Pictured above is what our tiny model predicts after "cheer cheer for old notre". It puts `26%` on "odds", `24%` on "be" and `22%` on "volley", and gives "dame" `0.08%`, **less than a thirtieth of what guessing evenly would give it** (`1/41`, or `2.4%`).

Counting Parameters

We now have every piece, so we can count GPT-2 small's weights. Attention has its four d × d matrices (W_Q, W_K, W_V and W_O), the feed-forward sublayer has W_1 and W_2, each of these comes with a bias, and every layer norm has a γ and a β:

```
V, T, d, L = 50257, 1024, 768, 12  # L blocks
attn = d * 3 * d + 3 * d + d * d + d   # WQ WK WV, WO
ffn = d * 4 * d + 4 * d + 4 * d * d + d   # W1, W2
norms = 2 * 2 * d                      # two layer norms
block = attn + ffn + norms
total = V * d + T * d + L * block + 2 * d
print(attn, ffn, block)
print(total)
```

OUTPUT

```
2362368 4722432 7087872
124439808
```

That's `124,439,808` weights: `38.6` million in the embeddings, `0.8` million in the positions and `85.1` million in the `12` blocks. Our tiny model has `1,096` (no biases, and no γ or β).

Notice that nothing in there is more exotic than a matrix multiply, an add, a normalization, a ReLU and a softmax. **It really is just linear algebra and statistics**, a lot of it.

The Forward Pass

Here is the whole model in PyTorch, from the ids of "cheer cheer for old notre" to the probabilities of "odds", "be", "volley" and "dame" coming next:

```
import numpy as np
import torch
def pos_enc(T, d):
    pos = torch.arange(T)[:, None]
    ang = pos / 10000 ** (torch.arange(0, d, 2) / d)
    P = torch.zeros(T, d)
    P[:, 0::2] = torch.sin(ang)
    P[:, 1::2] = torch.cos(ang)
    return P
def norm(X):                                    # layer norm
    mu = X.mean(1, keepdim=True)
    sd = ((X - mu) ** 2).mean(1, keepdim=True).sqrt()
    return (X - mu) / sd
V, d = 41, 8                                    # words, width
rng = np.random.default_rng(30124)
t = lambda *s: torch.tensor(rng.normal(*s),
                            dtype=torch.float32)
E = t(0, 1, (V, d))                             # embeddings
WQ, WK, WV, WO = t(0, d**-.5, (4, d, d))
W1 = t(0, d**-.5, (d, 4 * d))
W2 = t(0, (4 * d)**-.5, (4 * d, d))
x = [0, 0, 1, 2, 3]       # cheer cheer for old notre
T = len(x)
X = E[x] + pos_enc(T, d)
H = norm(X)
S = (H @ WQ) @ (H @ WK).T / d ** 0.5
mask = torch.triu(torch.ones(T, T), 1) == 1
S = S.masked_fill(mask, float("-inf"))      # mask
X = X + torch.softmax(S, 1) @ (H @ WV) @ WO  # attn
X = X + torch.relu(norm(X) @ W1) @ W2       # FFN
probs = torch.softmax(norm(X) @ E.T, 1)     # output
print(probs[-1, [24, 25, 14, 4]].round(decimals=4))
```

OUTPUT

```
tensor([0.2646, 0.2424, 0.2166, 0.0008])
```

Training a Transformer

Our model's guesses are bad because nothing has been learned, every weight is still what the random number generator gave it. To measure how bad, we use the same loss as logistic regression, **cross-entropy**, now over 41 classes instead of 2: the negative log of the probability the model gave the word that actually came next, averaged over every position.

cross-entropy over every position

$$[6.10] \quad \mathcal{L} = -\frac{1}{T-1}\sum_{t=1}^{T-1} \ln \hat{y}_{t,\,x_{t+1}}$$

Over the 50 guesses in the chorus our model's loss is 7.51. Guessing evenly would give $\ln 41 = 3.71$, so **our model is worse than knowing nothing**. It's confidently wrong: its favorite word gets 48% of the probability on average, and the favorite is right once in 50 tries (after "shake" it picks "down", by luck).

GPT-2 starts with much smaller weights (a standard deviation of 0.02), and an untrained model like that guesses close to evenly, a loss near $\ln 50257 = 10.8$. Training is then gradient descent on this loss with backpropagation, just as in the last few chapters, only with 124 million weights and billions of tokens.

Generating Text

Once a model is trained, generating text is a loop: run the forward pass, pick a word from the last row's probabilities, add it to the end of the input, and run it again. Picking the most likely word every time is called **greedy decoding**. Here's what greedy decoding does with our untrained model, starting from "cheer cheer for old notre":

> *cheer cheer for old notre odds odds odds odds volley volley volley volley*

Not quite the fight song. Real systems usually **sample from the probabilities** instead, so the same start can lead to different text, and they only look back as far as their context (`1,024` tokens for GPT-2).

Every new token costs a full pass through every block. That's fine for a paragraph of text. For a picture with hundreds of thousands of pixels it would be a lot of passes, which is part of why the models that generate images mostly don't work one piece at a time.