Transformers

Morgan Sinclaire

I thought of it as the eyes of Fourier, seeing God in space. Transformer, you have won. You are the accelerator of providence; your motions are the waves of causality. Time is self-similar through you. And yet...Who is writing this story?

– GPT-3

Outline

Outline

Tokens vs. Words

Technically, an LLM actually inputs *tokens*, not words (illustration). We discussed the details of this in `BPE.ipynb`.

Bottom line:

- For text in English or some other major languages, the tokens tend to be short/common words or at least morphemes.
- For rarer languages or non-language text, the tokens will often be UTF-8 bytes.

So "most of the time", it is fine to oversimplify and think of the tokens as words with discrete meanings.[1]

[1] Compare this with computer vision, where individual pixels/patches have no meaning and interpretability is harder.

Outline

Masked Self-Attention

Most 2020s LLMs are *generative*: they are given a sequence of tokens $t_1, \ldots, t_n$, and asked to predict the next token t_{n+1}.

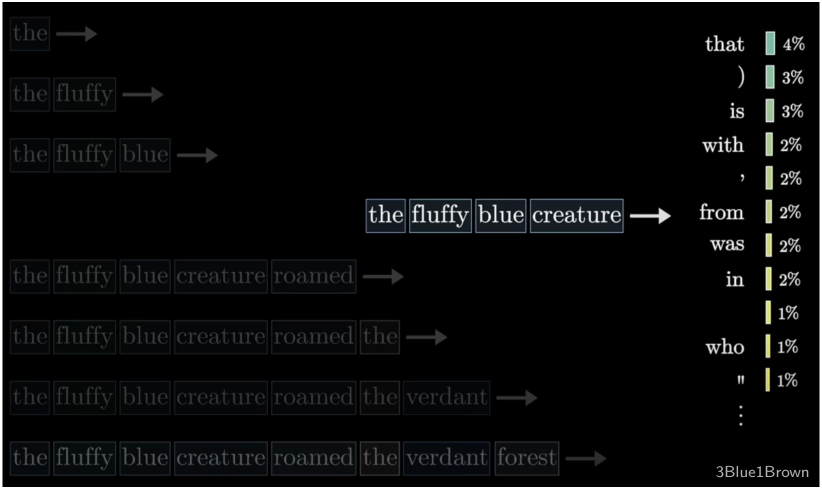
- Doing this repeatedly, the model then generates text.

The above is a bit oversimplified! When you feed it $t_1, \ldots, t_n$, it will actually calculate n predictions:

1. The second token t_2, given t_1.
2. The third token t_3, given t_1, t_2.
3. The fourth token t_4, given t_1, t_2, t_3.
4. $\vdots$
5. The $(n+1)$st token t_{n+1}, given $t_1, \ldots, t_n$.

Masked Self-Attention

This yields a lot more training examples:



the →
the fluffy →
the fluffy blue →
the fluffy blue creature →
the fluffy blue creature roamed →
the fluffy blue creature roamed the →
the fluffy blue creature roamed the verdant →
the fluffy blue creature roamed the verdant forest →
that 4%
) 3%
is 3%
with 2%
, 2%
from 2%
was 2%
in 2%
1%
who 1%
" 1%
⋮
3Blue1Brown

Masked Self-Attention

But there is a problem with our setup!

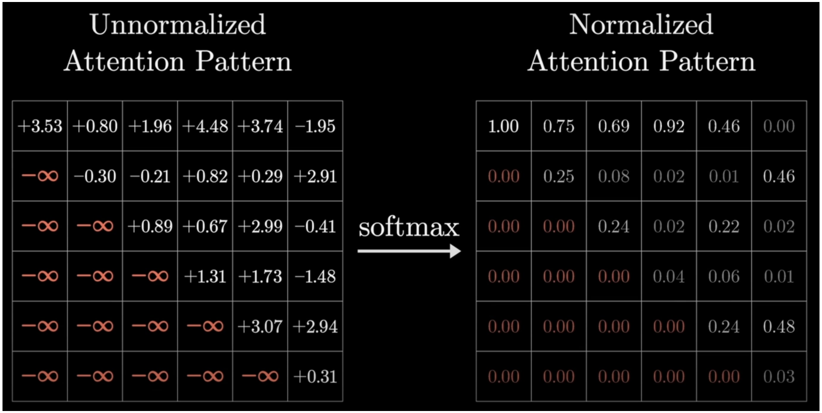
Input: A sequence of tokens $t_1, ..., t_n$.
Output: The sequence of tokens $t_1, ..., t_n, t_{n+1}$.

The prediction of the "next" token t_j is spoiled for every $j \leq n$.

We have to prevent the model from cheating by looking into the future.

Masked Self-Attention

Need to zero-out the future tokens:



Unnormalized Attention Pattern

+3.53	+0.80	+1.96	+4.48	+3.74	−1.95
−∞	−0.30	−0.21	+0.82	+0.29	+2.91
−∞	−∞	+0.89	+0.67	+2.99	−0.41
−∞	−∞	−∞	+1.31	+1.73	−1.48
−∞	−∞	−∞	−∞	+3.07	+2.94
−∞	−∞	−∞	−∞	−∞	+0.31

$\xrightarrow{\text{softmax}}$

Normalized Attention Pattern

1.00	0.75	0.69	0.92	0.46	0.00
0.00	0.25	0.08	0.02	0.01	0.46
0.00	0.00	0.24	0.02	0.22	0.02
0.00	0.00	0.00	0.04	0.06	0.01
0.00	0.00	0.00	0.00	0.24	0.48
0.00	0.00	0.00	0.00	0.00	0.03

We do this by setting the relevance scores to $-\infty$, so they get `softmax`'d to 0.

Outline

Logits

A generative transformer doesn't merely output the most probable token, it outputs a probability distribution over all possible tokens. GPT-3 has a vocab size of 50,257 distinct tokens it recognizes, so it outputs a probability distribution $p_1, p_2, \ldots, p_{50257} \in [0, 1]$ which sums to 1.

Actually, it first outputs the *logits* $\ell_1, \ell_2, \ldots, \ell_{50257} \in \mathbb{R}$; these then get softmax'd into probabilities in the final step.

The standard softmax function $S : \mathbb{R}^k \to \mathbb{R}^k$:

$$S([\ell_1, \ldots, \ell_k]) := \left[\frac{e^{\ell_1}}{\sum_{i=1}^{k} e^{\ell_i}}, \ldots, \frac{e^{\ell_k}}{\sum_{i=1}^{k} e^{\ell_i}} \right]$$

is just a general way of turning an arbitrary vector $\vec{\ell} \in \mathbb{R}^k$ into a probability distribution (even when $\vec{\ell}$ has negative values).

Softmax

Softmax S_T with a *temperature* $T \geq 0$:

$$S_T([\ell_1, ..., \ell_k]) := \left[\frac{e^{\ell_1/T}}{\sum_{i=1}^k e^{\ell_i/T}}, ..., \frac{e^{\ell_k/T}}{\sum_{i=1}^k e^{\ell_i/T}}\right]$$

<u>T large:</u>
As $T \to \infty$, this will flatten out to the uniform distribution.

<u>$T = 0$ (by which we mean the pointwise limit as $T \to 0^+$):</u>
This is just the $\arg\max()$ function, i.e. gives 1 to the highest and 0 to everything else.

Softmax

Setting temperature $T = 0$ the model will *always* output the most likely next token: the result will be bland, generic-looking text.

Setting temperature T larger and then randomly sampling from the flattened distribution: leads to more “spicy” completions.

Pretraining is done at $T = 1$, so these probs are privileged as the model's “true beliefs”. Adjusting T gives more/less “conservative” distributions at test-time:

- If we want rigid, reliable reasoning (e.g. in CoT prompting): set $T \leq .3$.
- If we want more creativity for brainstorming ideas etc.: set T higher.

Calibration

From GPT-4 tech report:

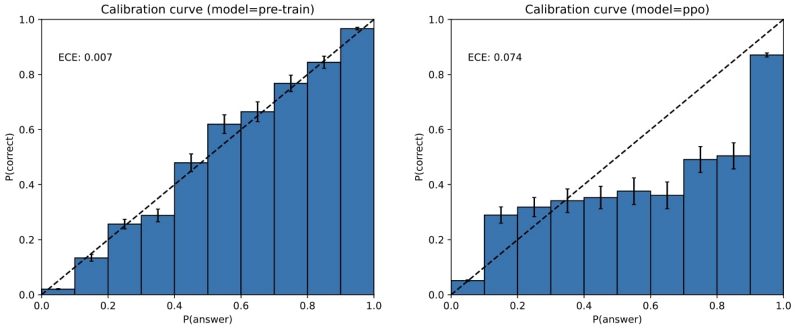


Figure 8. Left: Calibration plot of the pre-trained GPT-4 model on a subset of the MMLU dataset. On the x-axis are bins according to the model's confidence (logprob) in each of the A/B/C/D choices for each question; on the y-axis is the accuracy within each bin. The dotted diagonal line represents perfect calibration. Right: Calibration plot of the post-trained GPT-4 model on the same subset of MMLU. The post-training hurts calibration significantly.

The base model's next-token outputs form a well-calibrated probability distribution.
Sounding good to humans: need to be overconfident.

Outline

Why Cross Entropy?

We get a probability vector $Q = [q_1, ..., q_k]$ over the k tokens, and we find out that token i was correct. How do we score this in terms of q_i?

Need a differentiable function to do gradient descent (e.g. accuracy would lead to vanishing gradients).

Why not use absolute loss $:= 1 - q_i$?

Why Cross Entropy?

We get a probability vector $Q = [q_1, ..., q_k]$ over the k tokens, and we find out that token i was correct. How do we score this in terms of q_i?

Need a differentiable function to do gradient descent (e.g. accuracy would lead to vanishing gradients).

Why not use absolute loss $:= 1 - q_i$?

This would not be a *proper scoring rule*: can check that Q-expected loss is minimized by reporting extreme values (i.e. put probability 1 on most likely rather than true belief).

Log loss $:= -\log q_i$ can be shown to induce faithful reporting.

It is equivalent to *cross-entropy* from information theory.

Information Theory: Entropy

Let P be a discrete probability distribution over the sample space $\mathcal{X} = \{x_1, ..., x_k\}$.

Then $H(P) := \mathbb{E}[-\log P(x)] = -\sum_x P(x) \log P(x)$, the *entropy* of P, denotes the average $\#$ of bits needed to specify a draw from P.

<u>Example:</u> $H(\texttt{Unif}(1, ..., n)) = -\log \frac{1}{n} = \log n$, as expected.

For P non-uniform, we'd make Huffmann codewords $s_1, ..., s_k \in \{0,1\}^*$ for the outcomes based on frequencies. This would give us $|s_i| = \log \frac{1}{P(x_i)}$.

Then $H(P) = \mathbb{E}_P[|s_i|] = \mathbb{E}_P[-\log P(x)]$.

Information Theory: Cross-Entropy

Let $P, Q : \mathcal{X} \to [0,1]$, where P is the "true" distribution and Q is our model's estimate of it.

Then $H(P,Q) := \mathbb{E}_P[-\log Q(x)] = -\sum_x P(x) \log Q(x)$, is the *cross-entropy* of Q relative to P.

Interpretation: We want to describe outcomes from P in as few bits as possible, but instead we make codewords optimized for Q. How many bits will we use on average?

The best distribution to minimize these string lengths is P itself, so $H(P,Q)$ is a measure of "closeness" of P, Q.[2]

[2] The *Kullback-Leibler divergence* $D_{KL}(P \parallel Q) := H(P,Q) - H(P)$ is essentially the same thing since $H(P)$ is a fixed constant. As with D_{KL}, $H(P,Q) \neq H(Q,P)$ so these are not true metrics.

Log-loss = Cross-entropy

Setup:

- $X = \{x_1, ..., x_k\}$ possible labels.
- N samples $y_1, ..., y_N \in X$.
- Denote $Q :=$ the model's distribution.

Then average log-loss is:

$$\frac{-1}{N} \sum_{i=1}^{N} \log Q(y_i)$$

The training data defines our *empirical distribution* P by $P(x) := \frac{1}{N} \sum_{i=1}^{N} \mathbb{I}(y_i = x)$. Then the cross-entropy

$$H(P, Q) = E_P[-\log Q(x)]$$

gives the same sum.

Perplexity

Pretraining uses CE/LL, but for simpler explainability we use *perplexity*.

Suppose our log loss $\frac{-1}{N}\sum_{i=1}^{N}\log Q(y_i)$ is **4.087**.

Exponentiate: $2^{4.087} \approx \mathbf{17}$ is the average[3] value of $\frac{1}{Q(y_i)}$. This essentially means:

- Typically, we put **1/17** probability on the correct token.
- Our model gets the same loss as uniformly guessing among **17** tokens.

Perplexity simply cancels the log in this way:

$$PP := 2^{H(P,Q)} = 2^{\frac{-1}{N}\sum_{i=1}^{N}\log Q(y_i)} = \left(\prod_{i=1}^{N}\frac{1}{Q(y_i)}\right)^{1/N}$$

[3]This is a geometric mean across our dataset.

Outline

Embedding vs. Unembedding

Before the attention layers, the learned *embedding matrix* W_E computes the token embeddings.

GPT-3: W_E has size $12,888 \times 50,257$, since it maps the vocab size to the token embedding size.

Embedding vs. Unembedding

Before the attention layers, the learned *embedding matrix* W_E computes the token embeddings.

GPT-3: W_E has size $12,888 \times 50,257$, since it maps the vocab size to the token embedding size.

After the attention layers, the learned *unembedding matrix* W_U converts the updated embeddings into next-token logits over the possible tokens.

Hence in GPT-3, W_U has size $50,257 \times 12,288$.

(W_E, W_U always have transposed dimensions)

Embedding vs. Unembedding

Compare what the embedding/unembedding matrices do:

- W_E: tokens $\to$ vectorized meanings
- W_U: vectorized meanings $\to$ next-token probabilities

They are not literally "opposites", but there is some partial redundancy between them. For this reason a common procedure is *weight tying*, where we enforce $W_U = W_E^\top$ (often this is done without comment).

1. This saves on parameters.
2. Also seems to help with regularization.

Apparently this is less common in larger models, where W_E, W_U are only $\sim 1\%$ of the weights anyways.

Outline

Word vs. Sentence Embeddings

Bonus perspective: A transformer is an architecture for bootstrapping a word embedding into a sentence embedding.

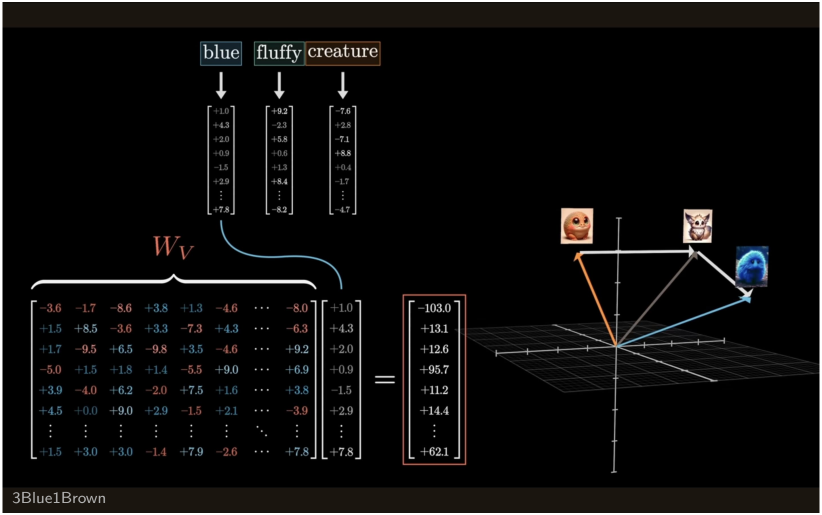
Consider the following problem:

Input: A sequence of tokens $t_1, \ldots, t_n$.
Output: An embedding vector $\vec{v}_{t_1,\ldots,t_n}$ that accurately captures the meaning of the text $t_1, \ldots, t_n$.

We know our embedding space is rich enough that some appropriate $\vec{v}_{t_1,\ldots,t_n}$ exists; how do we actually find it?

$E()$ is only designed to input individual tokens, we can't just feed it something else.

Word Embeddings → Sentence Embeddings



Have: $E(\texttt{blue})$, $E(\texttt{fluffy})$, $E(\texttt{creature})$ [shown in orange]
Want: $\overline{E}(\texttt{blue fluffy creature})$ [shown in blue]

Word Embeddings $\to$ Sentence Embeddings

Problem: Have a word embedding $E : \texttt{words} \to \mathbb{R}^d$, want to train a richer thought embedding $\overline{E} : \texttt{text} \to \mathbb{R}^d$.

Naïve idea 1: Take the average:

$$\overline{E}(w_1, ..., w_n) := \frac{E(w_1) + E(w_2) + ... + E(w_n)}{n}$$

- This does not work: "dog bites man" vs. "man bites dog".

Naïve idea 2: Train the same model on sentences instead of words.

- Combinatorial explosion of possibilities $\to$ combinatorial implosion of frequencies.
- GPT-3 has a *vocabulary size* of 50,257 words, meaning its *embedding matrix* is of size 12,288 x 50,527. One does not simply make this orders of magnitude larger.

Cumulative Meaning

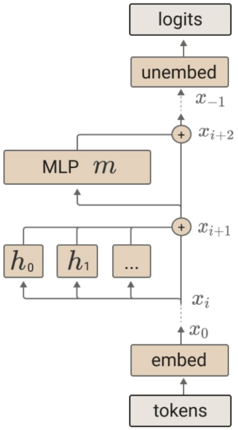
It's often said that a transformer computes the context-informed meaning of the tokens it takes in. This is not quite true!

It actually keeps track of the *cumulative meaning* of the tokens it has seen up to that point. In other words, on input $w_1, w_2, \dots, w_n$, its final attention block will output the embeddings:[4]

1. $\overline{E}(w_1)$
2. $\overline{E}(w_1, w_2)$
3. $\overline{E}(w_1, w_2, w_3)$
4. $\vdots$
5. $\overline{E}(w_1, w_2, \dots, w_n)$

[4] To see this, recall that its next-token prediction is obtained only from the last vector; this implies that *all* the contextual information needed has been fully represented in that vector by the end of the residual stream.

Cumulative Meaning



The final logits are produced by applying the unembedding.

$$T(t) = W_U x_{-1}$$

An MLP layer, m, is run and added to the residual stream.

$$x_{i+2} = x_{i+1} + m(x_{i+1})$$

Each attention head, h, is run and added to the residual stream.

$$x_{i+1} = x_i + \sum_{h \in H_i} h(x_i)$$

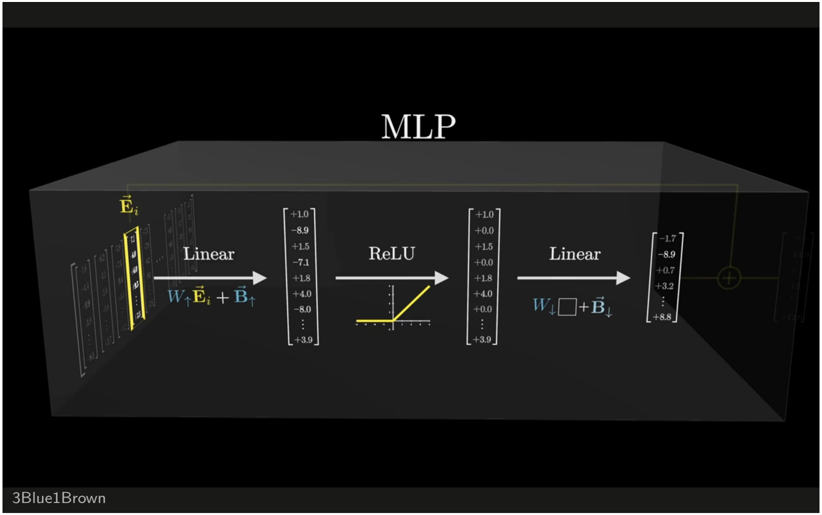
Token embedding.

$$x_0 = W_E t$$

Outline

MLP (Feedforward) Layers

Technicality: You typically use GELU (instead of ReLU).



The MLP layers are actually about $\sim 2/3$ of the parameters in GPT-3! What are they doing?

MLP Layers

The "dark matter" of LLM interpretability: what happens in these is even less understood than attention layers.

Weak and vague consensus: the attention layers handle "linguistic" information, the MLP layer is where it really "thinks/reflects" on the substance.

For example, as tokens enter an MLP layer, it might:

- Check for logical consistency.
- Cross-check with empirical facts it has learned.
- Learn new facts on the fly.[5]

[5]This can happen during a forward pass; called *in-context learning*.

MLP Layers

Let's look at an imaginative example. Suppose the input text is:

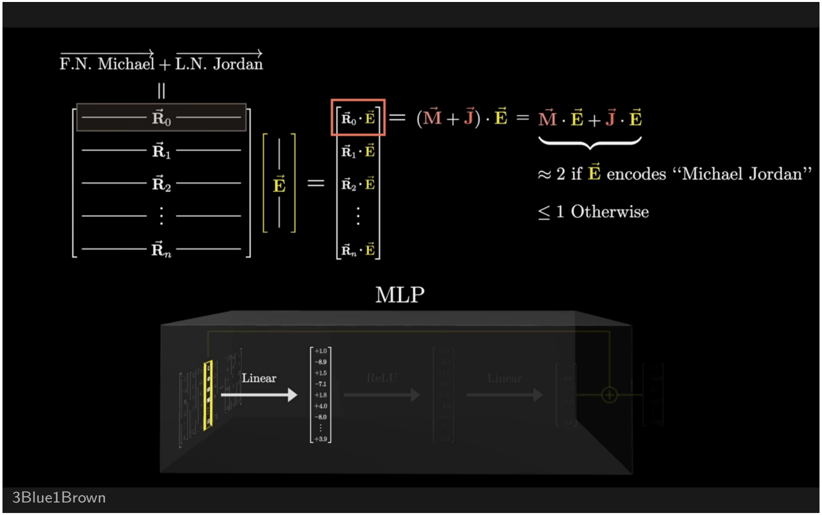
```
>>> Michael Jordan plays the sport of
```

The attention layers would've already computed the embeddings:

$$\vec{M} := \text{“}\texttt{First Name Michael}\text{”}$$
$$\vec{J} := \text{“}\texttt{Last Name Jordan}\text{”}$$

But we need to insert the learned empirical fact that *Michael Jordan plays basketball*.

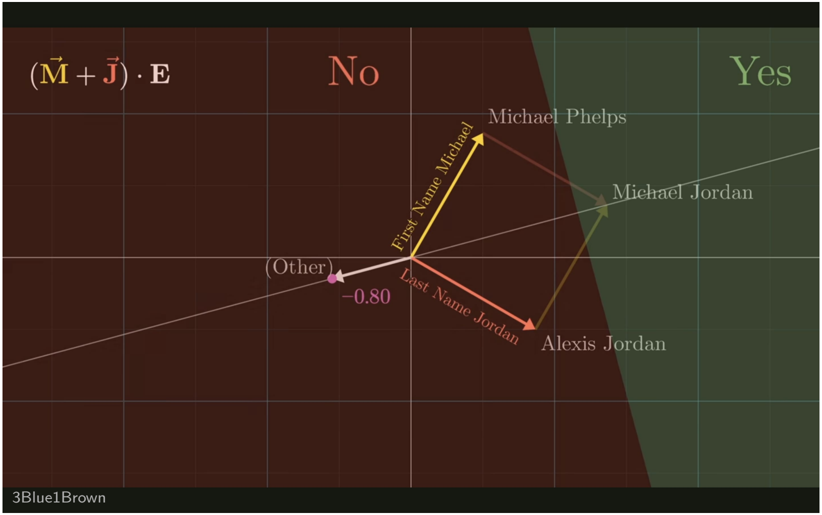
MLP Layers



$\overrightarrow{\text{F.N. Michael}} + \overrightarrow{\text{L.N. Jordan}}$
||
$\vec{R}_0$
$\vec{R}_1$
$\vec{R}_2$
$\vec{R}_n$
$\vec{E}$
$\vec{R}_0 \cdot \vec{E}$
$\vec{R}_1 \cdot \vec{E}$
$\vec{R}_2 \cdot \vec{E}$
$\vec{R}_n \cdot \vec{E}$
$= (\vec{M} + \vec{J}) \cdot \vec{E} = \underbrace{\vec{M} \cdot \vec{E} + \vec{J} \cdot \vec{E}}$
≈ 2 if $\vec{E}$ encodes "Michael Jordan"
≤ 1 Otherwise
MLP
Linear
+1.0
−8.9
+1.5
−7.1
+1.8
+4.0
−8.0
+3.9
ReLU
Linear
3Blue1Brown

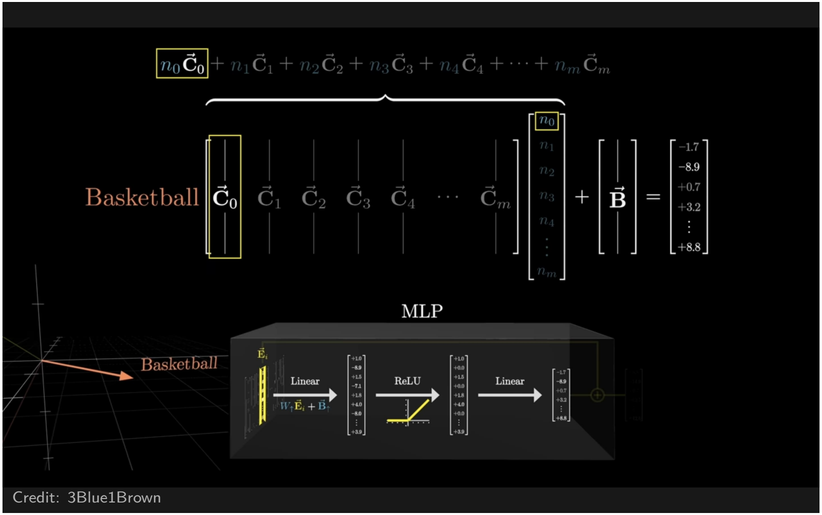
First linear layer: The rows $\vec{R}_i$ encode questions, e.g. $\vec{R}_0$ tests for the vector $\vec{M} + \vec{J}$.

MLP Layers



The nonlinear ReLU/GELU allows us to do an `AND` over $\vec{M}, \vec{J}$.

MLP Layers


$n_0\vec{C}_0 + n_1\vec{C}_1 + n_2\vec{C}_2 + n_3\vec{C}_3 + n_4\vec{C}_4 + \cdots + n_m\vec{C}_m$
Basketball
$\vec{C}_0$ $\vec{C}_1$ $\vec{C}_2$ $\vec{C}_3$ $\vec{C}_4$ $\cdots$ $\vec{C}_m$
n_0 n_1 n_2 n_3 n_4 n_m
$+ \vec{B} =$
−1.7 −8.9 +0.7 +3.2 +8.8
MLP
Basketball
Linear
$W_\uparrow\vec{E}_i + \vec{B}_\uparrow$
ReLU
Linear

Credit: 3Blue1Brown

Second linear layer: The columns $\vec{C}_j$ encode facts to be conditionally inserted, e.g. $\vec{C}_0$ adds the fact "plays basketball".

Outline

Polysemanticity

In neuroscience and DL, *monosemanticity* refers to the idea that neurons and features/concepts have a 1-1 correspondence.

This is cartoonishly wrong, but *if* it were true this would be great for interpretability!

In reality, for efficiency reasons, neurons are *polysemantic* and correspond to multiple features at once, but we can imagine trying to unfold it into a much larger monosemantic model that we can interpret better.

But *how* does polysemanticity happen?

Superposition

The *superposition hypothesis* is a proposed explanation for how neurons perform polysemanticity: it comes down to the counterintuitive geometric fact that in $\mathbb{R}^n$, we can only fit n mutually orthogonal vectors, but we can fit exponentially many near-orthogonal vectors (Johnson-Lindenstrauss lemma; see 17:04 of this video for a nice visualization).

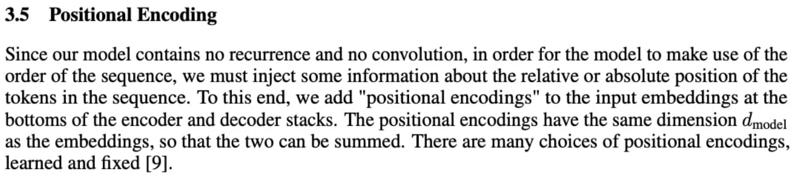
A major ongoing research program in LLM interpretability is to:

1. Test/investigate the superposition hypothesis.
2. Use it to do this unfolding into a more interpretable model, usually a *sparse autoencoder (SAE)*.
3. Break this down into individual circuits that can be pieced together in a reductionist manner.

Outline

Positional Encoding/Embedding

From "Attention Is All You Need":

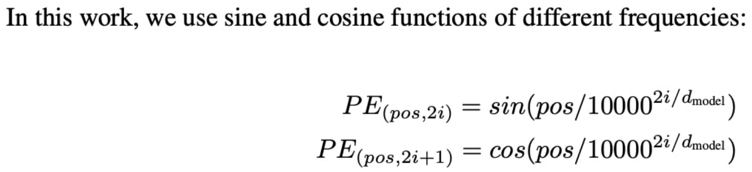

3.5 Positional Encoding

Since our model contains no recurrence and no convolution, in order for the model to make use of the order of the sequence, we must inject some information about the relative or absolute position of the tokens in the sequence. To this end, we add "positional encodings" to the input embeddings at the bottoms of the encoder and decoder stacks. The positional encodings have the same dimension d_{model} as the embeddings, so that the two can be summed. There are many choices of positional encodings, learned and fixed [9].

This happens between W_E and the first attention layer. The construction is a bit technical, but bottom line:

1. Without this, the transformer would be completely atemporal with no understanding of the ordering of its input tokens.
2. This is also the only place where the max context window is hard-coded (e.g. a 2049th token into GPT-3 would break the positional encoding).

Positional Encoding: Original Implementation



In this work, we use sine and cosine functions of different frequencies:

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$
$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$$

Yikes!

To motivate this, let's see why simpler ideas don't work as well.

Positional Encoding: The Setup

The t^{th} token has embedding $\vec{E}_t \in \mathbb{R}^{d_{model}}$. Want to compute positional vector $\vec{P}_t \in \mathbb{R}^{d_{model}}$ such that $\vec{E}_t + \vec{P}_t$ encodes both meaning and position.

Positional Encoding: The Setup

The t^{th} token has embedding $\vec{E}_t \in \mathbb{R}^{d_{model}}$. Want to compute positional vector $\vec{P}_t \in \mathbb{R}^{d_{model}}$ such that $\vec{E}_t + \vec{P}_t$ encodes both meaning and position.

First idea: Just define:

$$\vec{E}_t := [t, 0, 0, ..., 0]^\top$$

This does achieve injectivity/uniqueness, but it leads to unstably large values.

1 Dimension: Not Enough

Another 1-D idea: take a bounded, strictly increasing sequence like $a_t := 1 - \frac{1}{t}$, and define:

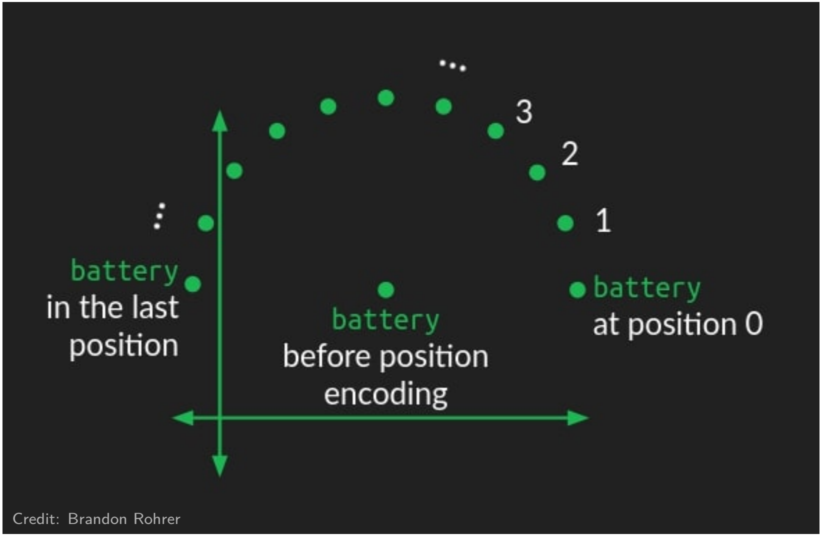
$$\vec{E}_t := [a_t, 0, 0, ..., 0]^\top$$

Problem 1: the larger positions get smushed together too closely.

Problem 2: The distance from $\vec{P}_1$ to $\vec{P}_2$ is way different from (say) $\vec{P}_{1000}$ to $\vec{P}_{1001}$. This confuses the model; better to make things evenly spaced.

Upshot: 1 dimension doesn't have enough room to make this work well.

Even Spacing



For thousands of tokens, a simple circle still doesn't have enough room. But with more dimensions, it points at the right idea.

Analogy: Clocks

The 3 hands of a clock move cyclically with different periods. Together, they point to a unique time of day.

We can parameterize their motions as:

$$\begin{aligned}
\text{Second hand:}\ & (\cos(2\pi t), && \sin(2\pi t)) \\
\text{Minute hand:}\ & (\cos(2\pi t/60), && \sin(2\pi t/60)) \\
\text{Hour hand:}\ & (\cos(2\pi t/720), && \sin(2\pi t/720))
\end{aligned}$$

The “max context window here” is 12 hours, after which uniqueness breaks down.

Analogy: Clocks

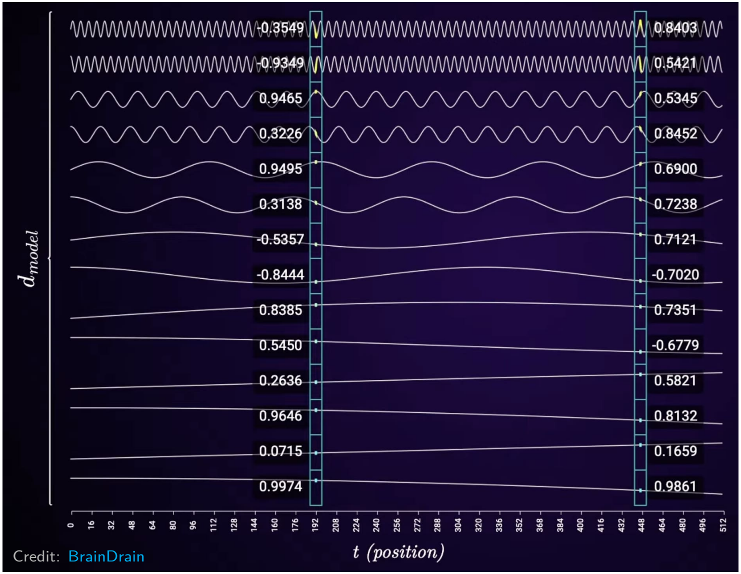
We can package this into a single position vector:[6]

$$\begin{bmatrix} \sin(2\pi t) \\ \cos(2\pi t) \\ \sin(2\pi t/60) \\ \cos(2\pi t/60) \\ \sin(2\pi t/720) \\ \cos(2\pi t/720) \end{bmatrix}$$

Add thousands more "hands" and this is *almost* it!

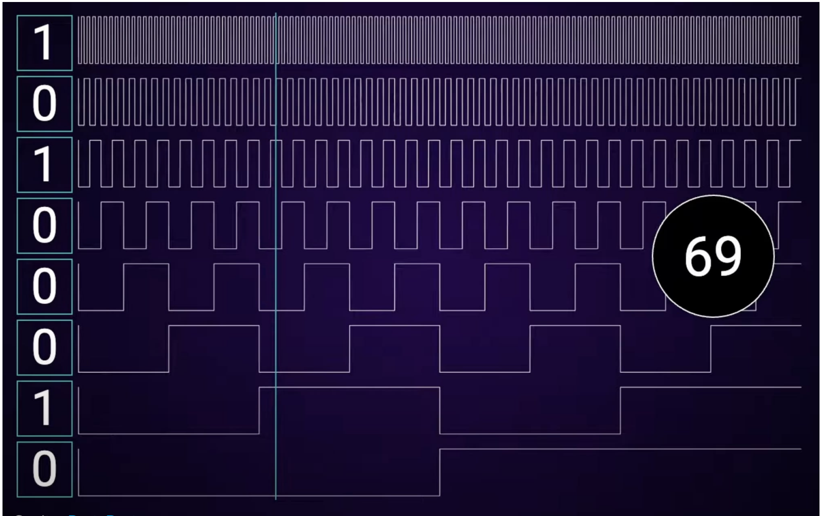
[6] Possible visualization: a point on a torus travels around the major circle every hour, around the minor circle every minute, and around orthogonal "epicycles" in other dimensions on other frequencies. Or this.

Frequencies



Each pair of rows is a hand of a clock, each column is a unique $\vec{P}_t$.

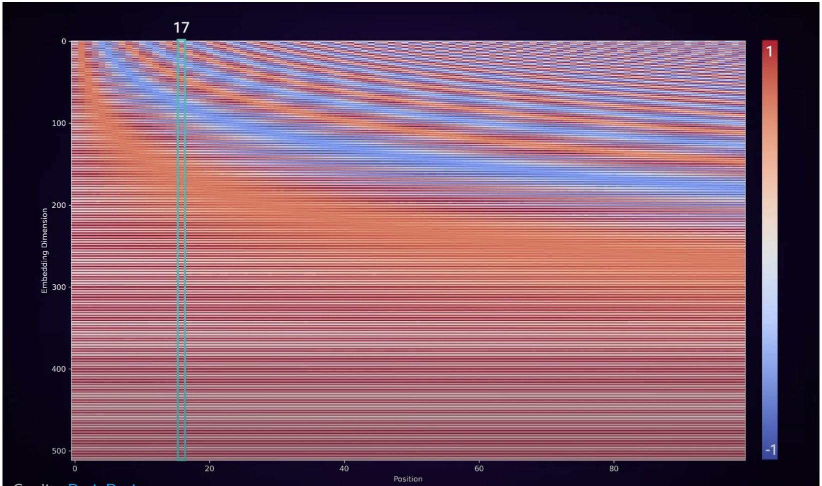
Discrete Analogy: Binary Numbers



Credit: BrainDrain

Each horizontal line corresponds to a unique value.
Sinusoids are just the floating-point version of this.

Frequency Map



Credit: BrainDrain

Each pair of rows is a hand of a clock, each column is a unique $\vec{P}_t$.

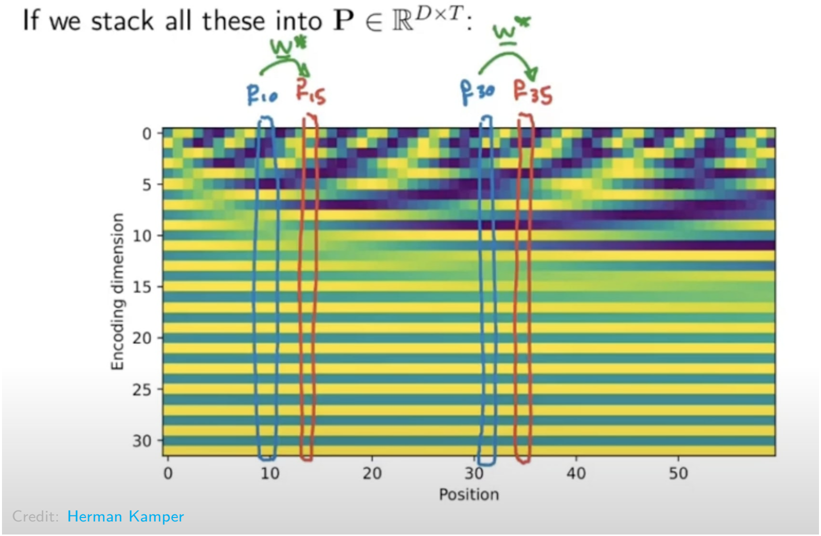
Computing $\vec{P}_t$

$$\vec{P}_t := \begin{bmatrix} \sin(t) \\ \cos(t) \\ \sin(t/N^{2/d}) \\ \cos(t/N^{2/d}) \\ \sin(t/N^{4/d}) \\ \cos(t/N^{4/d}) \\ \vdots \\ \sin(t/N^{(d-2)/d}) \\ \cos(t/N^{(d-2)/d}) \end{bmatrix} \in \mathbb{R}^{d_{model}}$$

where $N > d_{model}$ is a hyperparameter. Setting $N = 10000$, this can be written:

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$
$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$$

Relative Positioning

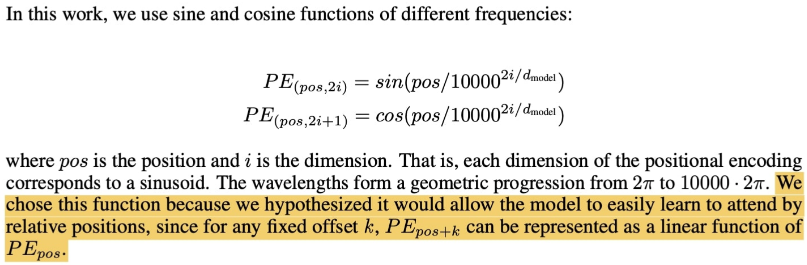


Credit: Herman Kamper

For any k, there is a specific matrix W s.t. $W \cdot \vec{P}_t = W \cdot \vec{P}_{t+k}$.
See here for a proof.

Relative Positioning

This is important! The model is made of linear maps, so now it has an easy way to track/manipulate relative positions.



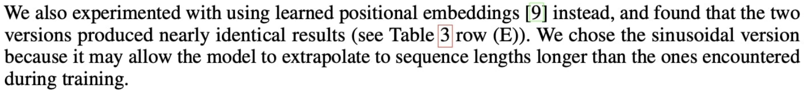
In this work, we use sine and cosine functions of different frequencies:

$$PE_{(pos,2i)} = sin(pos/10000^{2i/d_{\text{model}}})$$
$$PE_{(pos,2i+1)} = cos(pos/10000^{2i/d_{\text{model}}})$$

where pos is the position and i is the dimension. That is, each dimension of the positional encoding corresponds to a sinusoid. The wavelengths form a geometric progression from 2π to $10000 \cdot 2\pi$. We chose this function because we hypothesized it would allow the model to easily learn to attend by relative positions, since for any fixed offset k, PE_{pos+k} can be represented as a linear function of PE_{pos}.

Learned Positional Encodings

This option is also done in practice sometimes:


We also experimented with using learned positional embeddings [9] instead, and found that the two versions produced nearly identical results (see Table 3 row (E)). We chose the sinusoidal version because it may allow the model to extrapolate to sequence lengths longer than the ones encountered during training.

Rotary Positional Embedding (RoPE)

Many other PE schemes have been proposed since 2017. RoPE (2023) is the hot new one (used in e.g. the DeepSeek models). Some key points:

1. Moves away from the "$\vec{E}_t + \vec{P}_t$" framework: instead rotates $\vec{E}_t$ directly.
2. Focuses more directly on *relative* positions rather than *absolute* positions like $PE(\vec{E}_t, t)$.
3. Interacts with the attention mechanism in a nontrivial way: computes separate PEs for query and key vectors.

Outline

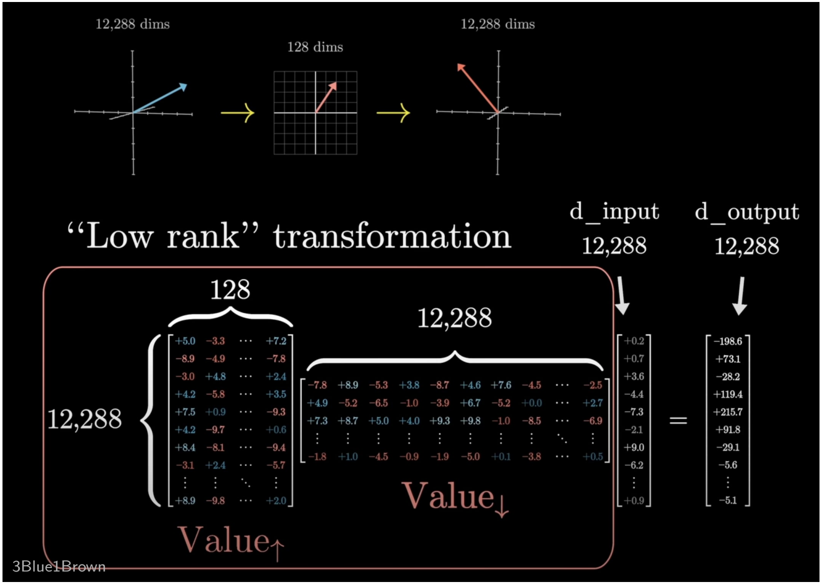
Value/Output Matrices

Recall that in GPT-3, the K, Q matrices have dimension $128 \times 12,288$.

As we described it, the V matrix would have dimension $12,288 \times 12,288$. This is way too big!

For efficiency reasons, V is factored into a *low-rank approximation* of two smaller matrices, of size $12,288 \times 128$ and $128 \times 12,288$. Let's call these $V_{\uparrow}$ and $V_{\downarrow}$.

Value/Output Matrices



Let's call these $V_{\downarrow}^{h}$ (down-projection) and $V_{\uparrow}^{h}$ (up-projection).

Value/Output Matrices

The $V_{\uparrow}^{h}$'s for each attention head h in a given layer are concatenated together to form the *output matrix*:

$$W_O := \left[V_{\uparrow}^{h_1} \mid V_{\uparrow}^{h_2} \mid V_{\uparrow}^{h_3} \mid \ldots \mid V_{\uparrow}^{h_{\text{num\_heads}}} \right]$$

(This is mathematically equivalent, it's just more efficient to do one big matmul than lots of small ones.)

Similarly, the $V_{\downarrow}^{h}$'s are also usually stitched together, into what's actually called the *value matrix* W_V.[7]

[7] Actually, this stitching is also done for query/key matrices, it's just less intuitive and not often depicted in explainers/diagrams.

Outline

RMSNorm

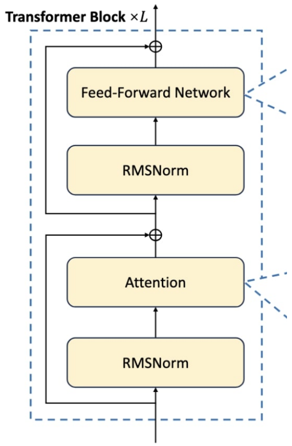


Figure 1: From DeepSeek-V3 paper

It's typical in DL to normalize input data in some way.

Early transformers used LayerNorm, where we simply normalize the activations in a given layer to have mean $\mu = 0$ and variance $\sigma^2 = 1$.

Modern LLMs use RMSNorm, which essentially achieves the same effect in a more hacky/efficient way.

Outline

Full Architecture

GPT-3 on a napkin

Outline

Marr's 3 Levels of Analysis

When analyzing a computational system, there are 3 levels at which to analyze it:

1. **Computational level**: What problem(s) is the system solving?
2. **Algorithmic level**: What methods/processes is the system using to solve (1)?
3. **Physical/implementation level**: How, engineering-wise, is (2) physically realized?

This framework applies to *any* system (Marr's original example was a cash register).

Marr's 3 Levels of Analysis

E.g. the human brain (as we understand it):[8]

1. **Computational level**: Minimize prediction error in continuous sensory data to make quick decisions, while respecting constraints such as energy efficiency.
2. **Algorithmic level**: In a hierarchy of predictive systems, sense flows up, predictions flow down, and intermediate levels resolve the conflicts based on relative strength.
3. **Physical/implementation level**: Carbon-based biological neurons, which communicate via rate/timing of electrical signals over ion channels.

[8] According to predictive processing theory, which is lately the main paradigm in neuroscience.

Marr's 3 Levels of Analysis

A generative transformer:

1. **Computational level**: Next-token prediction on (its subsample of) internet data.
2. **Algorithmic level**: Attention algorithm: use KQ pairs to compute relevance, use this and V to compute cumulative meaning of entire text. Also, feedforward layers for nonlinguistic thinking/reflection. Gradient descent to correct errors in loss function.
3. **Physical/implementation level**: Uses matrices of floating-point numbers on GPUs, representing KQV scores as individual row/column vectors to be multiplied. LOTS of low-level black magic hacks to make the H100s sing in chorus.

Are transformers the unique secret to intelligence?

No. As far as we can tell, RNNs had similar returns to scale. Language models weren't worth scaling up until the late 2010s.

Alternate timeline: transformer never invented. GPT-2 comes out in the 2020s as a char-RNN. RNNs hailed as "the universal architecture of intelligence".

Another timeline: RNNs never invented. GPT-2 comes out in the 2030s as a feedforward NN trained on next-char prediction. Feedforward NNs hailed as [...]

The actual star of the story: training on a general-purpose task (e.g. next-token prediction) with trillions of data points.

Language Models: Large and In Charge

Model	Year	Params	Corpus Size	FLOP	Cost
GPT-1	2018	0.117B	1e9	1.8e19	?
GPT-2	2019	1.5B	8e9	1.9e21	$\sim$ $50K
GPT-3	2020	175B	3.7e11	3.1e23	$2.1M
GPT-4	2023	$\sim$ 1T	?	1-4e25	$\sim$ $50M

Reference: Epoch AI. Many figures are estimates.

Fun fact: OpenAI incorrectly calculated the model sizes for the GPT-2 subfamily, and their original paper is wrong (1, 2).

LLM Dimensionalities

Standard notation for dimensionality hyperparameters:

d_{model}	size of token embedding space
d_{head}	size of query/key/value vectors
d_{ff}	size of feedforward/MLP activation space ("neurons")
n_{layer}	# of attention layers
n_{heads}	# of parallel heads per layer
n_{ctx}	max context size
d_{vocab}	BPE vocab size

LLM Dimensionalities

	d_{model}	d_{head}	d_{ff}	n_{heads}	n_{layer}	n_{ctx}	d_{vocab}
GPT-1	768	64	3072	12	12	512	40478
GPT-2 small	768	64	3072	12	12	1024	50257
GPT-2	1600	64	6400	25	48	1024	50257
GPT-3	12288	128	49152	96	96	2048	50257
DeepSeek-V3	7168	128	18432[9]	128	61	128k	128k

Common conventions:

1. $d_{key} = d_{value}$ (i.e. d_{head} is well-defined)
2. $d_{model} = n_{heads} \cdot d_{head}$ (orth. decompose the embedding space)
3. $d_{ff} = 4 \cdot d_{model}$

[9]Uses MoE with 1 shared expert and 8 routed experts (chosen from 256). Each has dimension 2048, so effectively $d_{ff} = (1 + 8) \cdot 2048 = 18432$.

Are transformers emulating the human brain?

No. There's a *very loose* analogy to be made:

CNNs : the visual cortex

RNNs : the rest of the brain

In particular, RNNs have recurrent (self-looping) connections, like most of the brain.[10]

In contrast, a transformer is completely acyclic (i.e. a DAG); RNNs $\rightarrow$ transformers was a step *away* from biological inspiration.

The DAG-ness is what allows more parallelization in training; essentially, it's an efficiency hack to more quickly do what an (attention-based) RNN already could (evolution wasn't clever enough to make this hack).

This should not be surprising! E.g. we didn't get flying machines by closely emulating bird wings; the results are over 100x faster.

[10] This was thought to be necessary for "thinking"; turns out a transformer's MLP layers suffice.

What's the actual secret to intelligence? (Opinion)

The bitter/unsatisfying lesson:

A massive, efficient search over the space of Turing machines[11]

Scaling gives us that:

- Model size: need enough capacity to store most of the useful TMs.
- Dataset size: need a well-trained prior to conduct the search.
- Inference-time compute: need more "juice" to conduct the search.

[11] Each TM represents an "idea"/"action"/"procedure" that may be useful in some situations.

What's the actual secret to intelligence? (Opinion)

The bitter/unsatisfying lesson:

A massive, efficient search over the space of Turing machines[11]

Scaling gives us that:

- Model size: need enough capacity to store most of the useful TMs.
- Dataset size: need a well-trained prior to conduct the search.
- Inference-time compute: need more "juice" to conduct the search.

But what about creativity?

[11] Each TM represents an "idea"/"action"/"procedure" that may be useful in some situations.