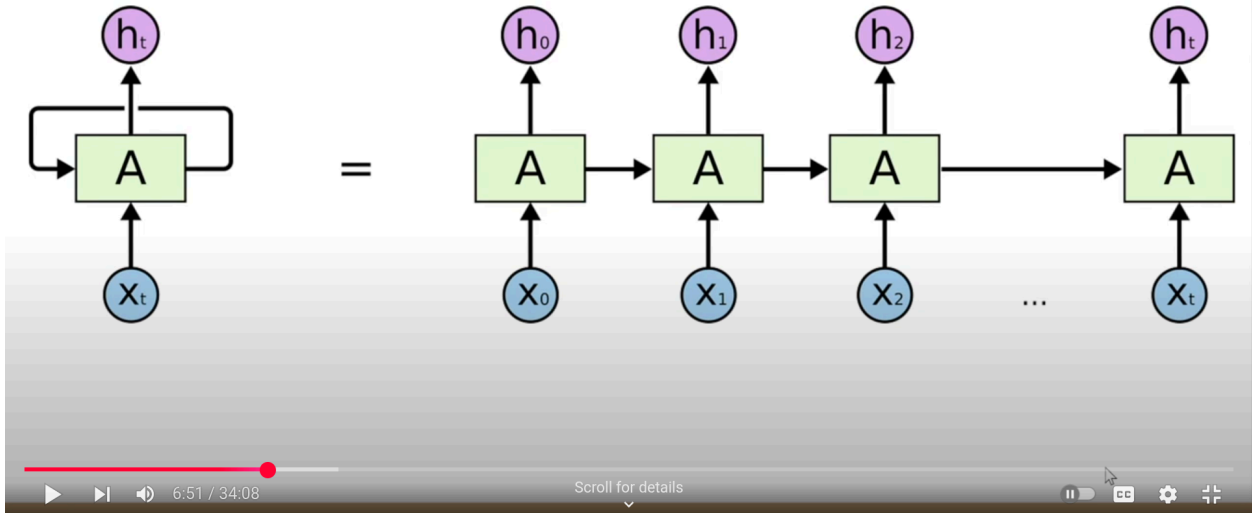
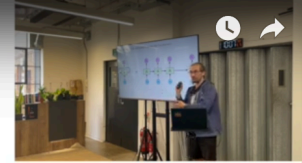


# Induction Heads

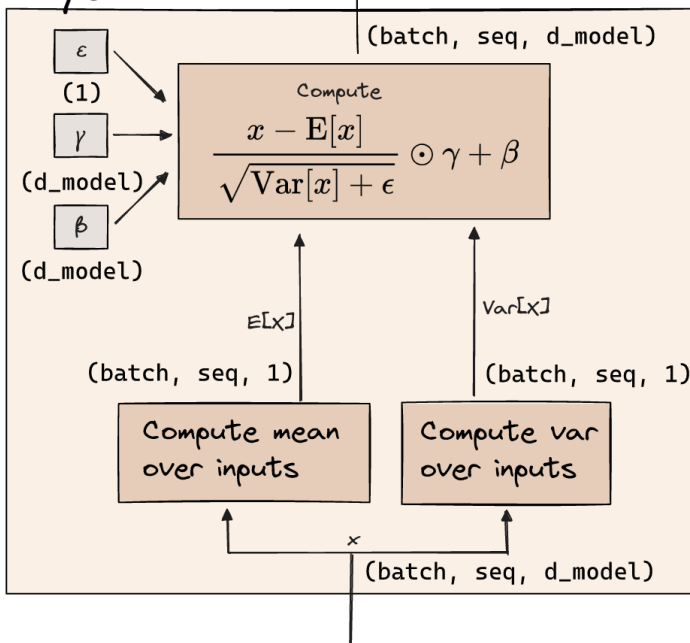
- RNNs can't be run/trained in parallel like Transformers



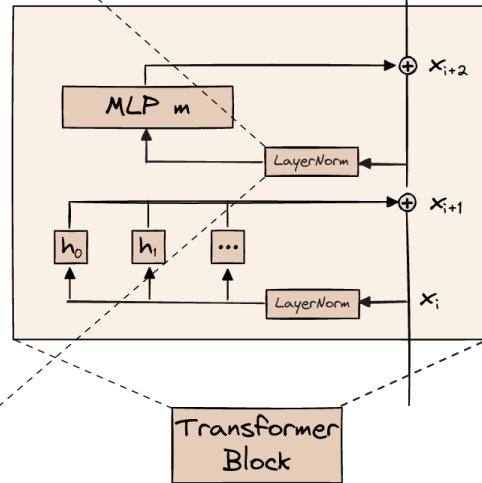
Transformers no need to process tokens sequentially like hidden state RNNs

Pushing residual stream can be pushed to drive model behaviour. Different blocks read and write from it. U can read from the residual stream in advance and try to decode what is going to be the final outcome.

# LayerNorm (LN)



# Transformer Block



Thought for 13s

Great “why” to ask. The short version: **residuals make very-deep nets easy to optimize and safer to use.** Concretely, in a Transformer we keep a single “residual stream”  $x$  and let sublayers write *updates* to it:

$$y = x + F(\text{LN}(x)) \quad (\text{attention or MLP})$$

### 1. Identity path $\Rightarrow$ stable gradients

Backprop through  $y = x + F(x)$  gives

$$\frac{\partial y}{\partial x} = I + J_F(x).$$

The  $I$  term means gradients can flow even if  $J_F$  is small/poorly conditioned. This largely prevents vanishing gradients and lets you stack many layers.

### 2. Easy “do nothing” option

If a block isn’t helpful yet, it can learn  $F(\cdot) \approx 0$  (or start near zero at init). Then  $y \approx x$ . Without a skip, the block must learn the **identity map** exactly, which is surprisingly hard and destabilizes training.

### 3. Incremental refinement

Each block learns a *delta* or **residual update** to the shared representation, rather than rewriting it from scratch. That makes learning smoother (“keep what works, add a tweak”).

### 4. Information highway

The raw token features can flow across many layers untouched via the skip, while attention/MLP inject new info. Without the skip, useful features can get overwritten or drift each layer.

### 5. Pre-LN + residual = well-behaved scales

LN before the sublayer gives  $F$  a normalized input; the skip keeps the base scale of  $x$ . Together this keeps activations in a good range and reduces the need for tricks.

### 6. Modularity of writers

In Transformers, **attention** mixes tokens (sequence axis) and **MLP** mixes channels (feature axis). Both write their updates into the same residual stream via **addition**, so the next block sees a unified representation.

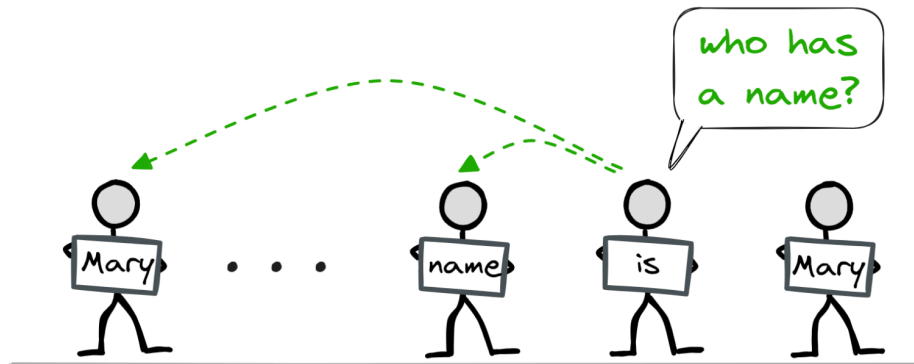
Tiny toy intuition (row-wise): if a block outputs zeros,

- **with skip:**  $y = x$  and  $\frac{\partial y}{\partial x} = I \rightarrow$  gradients pass cleanly.
- **without skip:**  $y = 0$  and  $\frac{\partial y}{\partial x} = 0 \rightarrow$  gradients die.

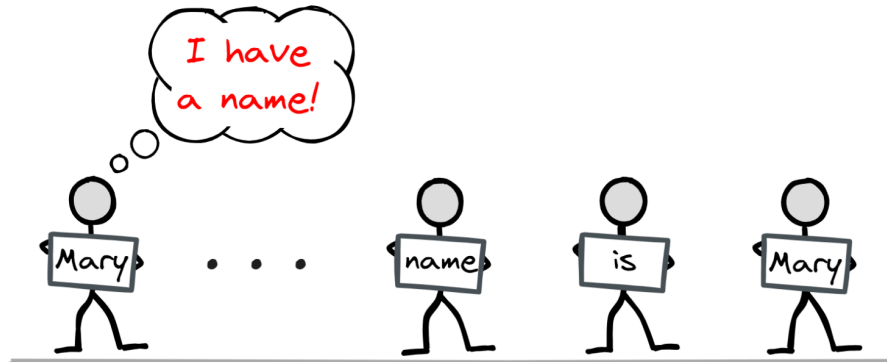
Engineering trick to add not transform is due to prevent vanishing gradients if  $F(\cdot) = 0$   
Then gradient can be 1, if  $F(\cdot) = 0$ , then gradient will be zero

Also , hand waving explanation: add to stream, instead of manipulating

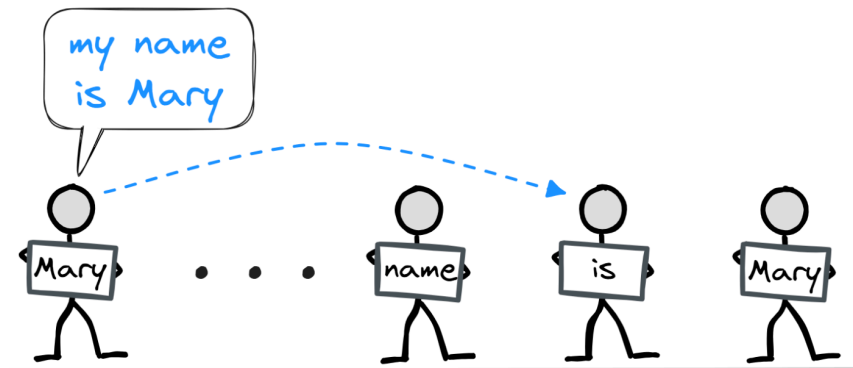
**Query**  
what question  
is asked



**Key**  
who chooses  
to reply



**Value**  
what info gets  
sent back to  
the asker



Thought for 50s

Yep — that einsum is where the heads get combined.

What it's doing exactly:

- For each head  $h$ :  
 $O_h = V_h W_O[h]$  where  $V_h \in \mathbb{R}^{B \times N \times e}$  and  $W_O[h] \in \mathbb{R}^{e \times d}$ .
- Then it **sums over heads**:  
 $\text{output} = \sum_{h=1}^H O_h \in \mathbb{R}^{B \times N \times d}$ .
- Finally it adds the output bias  $b_O$ : `+ rearrange(self.b_o, 'd -> 1 1 d')`.

So it's not a "plain average"; each head is first linearly projected by its own  $W_O[h]$ , and those **projected contributions are added**. This is **mathematically equivalent** to the usual "concat heads then multiply by a single  $W_O$ " view:

python

Copy Edit

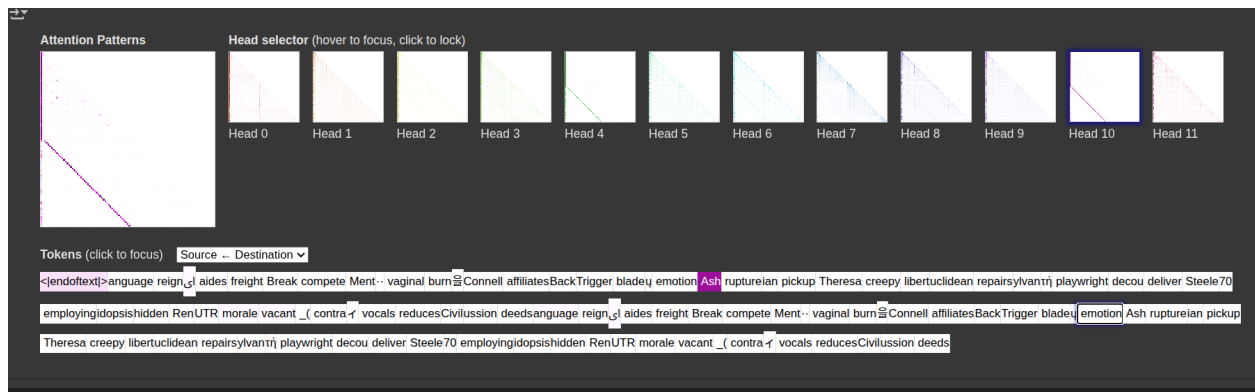
Each head, output matrix is obtained, and then added over all heads

Transformer figures

[https://raw.githubusercontent.com/info-arena/ARENA\\_img/main/misc/full-merm.svg](https://raw.githubusercontent.com/info-arena/ARENA_img/main/misc/full-merm.svg)

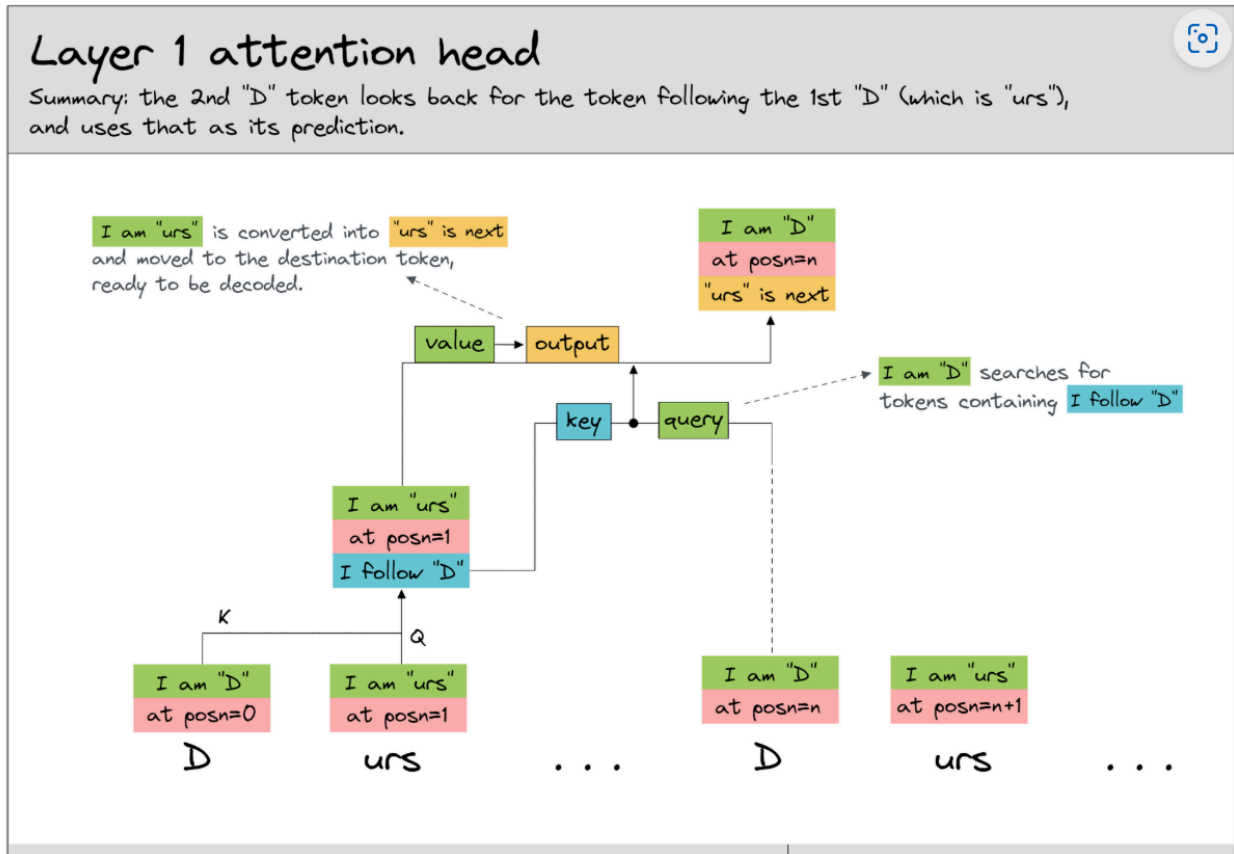
[https://raw.githubusercontent.com/info-arena/ARENA\\_img/main/misc/small-merm.svg](https://raw.githubusercontent.com/info-arena/ARENA_img/main/misc/small-merm.svg)

## Induction heads



This is an induction head at second layer. This attention head at "emotion" word, looks back at the previous occurrence of "emotion" and attends to what comes after *previous* "emotion". Since "Ash" comes after "emotion" previous case, it("emotion" token) is attending to "Ash" to use that as prediction for what comes after it("emotion" token)

## Explanation in less wrong



More concrete explanation on what query and key values pairs do:

## QK circuit

The query is  $\begin{matrix} \text{I am "D"} \\ \text{at posn=n} \end{matrix}^T W_Q = \text{"I'm looking for a token following "D" "}$

The key is  $\begin{matrix} \text{I am "urs"} \\ \text{at posn=1} \\ \text{I follow "D"} \end{matrix}^T W_K = \text{"I follow "D" "}$

The attention score (the dot product of the key and query vectors) is large, because the key is a good match for the query.

### What about the first head?

The first head makes it possible.

If the attention head in the first layer attends to key "I follow emotion".

The 2nd occurrence of "emotion" has a query "Who follows emotion"

The first occurrence of "Ash" ( the word that comes after "emotion" in first occurrence" ) has the key "i come after emotion"

This "KEY" information is added to the residual stream by the first head.

The 2nd head is able to read out from the residual stream and hence it attends to "Ash"

GPT explanation:

*Great question — the "mystery job" of the first head is what makes the second-layer induction jump even possible.*

*Here's the core idea, in plain terms:*

**What the first-layer head does (the "previous-token head")**

At each position  $t$ , it **looks one step back** (attends to  $t-1$ ) and then **writes into position  $t$  a tag that says “I follow token  $X$ ”**, where  $X$  is the token at  $t-1$ . Concretely, it writes a vector in the residual stream at position  $t$  that behaves like the **key** for token  $X$ . So every position gets labeled with “who I follow.” This is often called **K-composition**: the first head writes something **key-like**; a later head will use it when computing attention

## OV circuit - why is it diagonal

What is OV circuit:

If the input sequence is A,B,A, ?

Say Vocabulary is A,C,B (not ordered so that sequence and vocabulary are not confused)

The next predicted

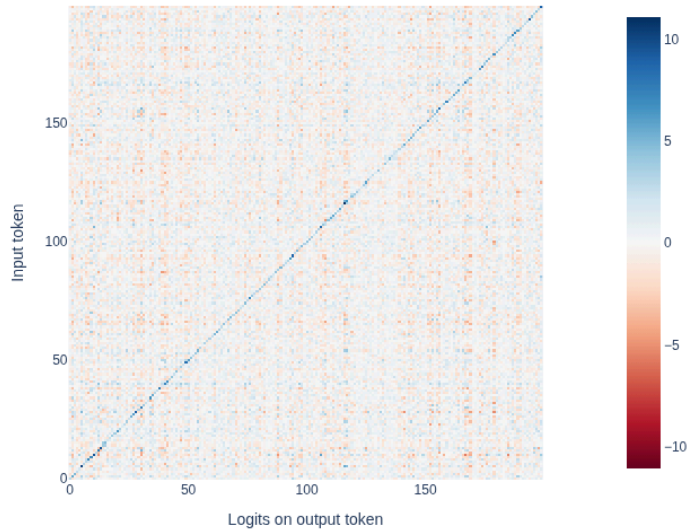
OV circuit is in the 2nd layer’s attention head that causes the output to B. This happens possible because the head in 1st layer passes information via residual stream information so that the in the attention matrix’s 2nd layer’s head,, second occurrence of “A” attends to previous occurrence of “B”. Put more generally, “A” attends to the token that it was previously followed by(Previously B followed A, so A attends to that previous B)



Example here( current “break” attends to word after previous occurrence of “break” which is “compet”)

Now OV circuit when plotted looks diagonal

Full OV circuit for copying head



Why does it happen?

First we break down it as the matrices

OV circuit from embedding to unembedding would be

$W_U W_O W_V W_E$

The embedding matrix  $W_E$  is multiplied with input sequence  $X$  ( $d_{\text{vocab}} \times \text{Seq\_len}$ )

And then that  $W_E @ X$  is multiplied with Attention Matrix  $A$  ( $\text{Seq\_len} \times \text{Seq\_len}$ ), and then these matrices  $W_V$  and  $W_O$  and  $W_U$  are multiplied in that order

So u can write this operations as

$W_U W_O W_V W_E X A$  ( to see it more clearly  $W_E$  times  $X$  is input in  $d_{\text{model}}$  dimensional space and then u multiply the input with attention matrix)

Now thinking about the sizes

$W_U$ :  $d_{\text{vocab}} \times d_{\text{model}}$

$W_O$ :  $d_{\text{model}} \times d_{\text{head}}$

$W_V$ :  $d_{\text{head}} \times d_{\text{model}}$

$W_E$ :  $d_{\text{model}} \times d_{\text{vocab}}$

$X$ :  $d_{\text{vocab}} \times \text{Seq\_len}$

$A$ :  $\text{Seq\_len} \times \text{Seq\_len}$

So, the OV matrix( $W_U W_O W_V W_E$ ) is of size  $d_{\text{vocab}} \times d_{\text{vocab}}$

And the input is  $XA$  ( $d_{\text{vocab}} \times \text{Seq\_len}$ )

So, the output logits has to be  $d_{\text{vocab}} \times \text{Seq\_len}$

We are interested in the last token prediction so we want to focus on last column of Logits

Matrix:  $\text{Logits}(:, -1)$

In  $\text{Logits}(:, -1)$ , we want the correct token index to be maximum. We can call it

$\text{correct\_token\_idx}$

So  $\text{Logits}(\text{correct\_token\_idx}, -1)$  has to be high

But  $\text{Logits}(\text{correct\_token\_idx}, -1) = \text{Dot product}(\text{OV}(\text{correct\_token\_idx}, :), \text{Input}(:, -1))$

We know Input is  $XA$  and we saw from attention matrices that the attention matrix attends mainly to the correct next token in the past. So,  $\text{Input}(\text{correct\_token\_idx}, -1)$  is high

So  $\text{Logits}(\text{correct\_token\_idx}, -1) = \text{product of terms } \text{OV}(\text{correct\_token\_idx}, \text{correct\_token\_idx})$  and  $\text{Input}(\text{correct\_token\_idx}, -1)$

To make  $\text{Logits}(\text{correct\_token\_idx}, -1)$  high in situations where  $\text{Input}(\text{correct\_token\_idx}, -1)$  is high,  $\text{OV}(\text{correct\_token\_idx}, \text{correct\_token\_idx})$  has to be high.

Hence OV circuit matrix is mainly diagonal

Key take aways:

- The rewriting of the OV circuit matrix with input as keep attention on right side. Naively, it is easy to think the operation as  $W_U W_O W_V \mathbf{A} W_E \mathbf{X}$ , but one can write it as  $W_U W_O W_V (W_E \mathbf{X}) \mathbf{A}$   
This is a simple, clever way. One nice way to see it is that operations on input space and feature space are separated. Operations on Input space involve  $X, A$  matrices (they have sequence length in their matrix dimension). So you can put them on the right corner
- Learning to write matrix products as summation / einstein notation is very important.

## How should full K-circuit look like?

# Superposition

# Overcomplete basis

To represent  $n$  vector, normally you would want “ $n$ ” basis. Ex: 1,0 ; 0,1

But overcomplete basis is when you have more than  $n$  basis. Ex: 1,0 ; 0,1 ; 0.5,0.5

So now you can have multiple sets of coefficient. To solve this problem, we introduce sparsity where the goal is most of the coefficients are zero.

Sparsity is energy efficient because most of the time inputs are sparse, you don't need many basis to be active. Only with few active basis would suffice.

<http://ufldl.stanford.edu/tutorial/unsupervised/SparseCoding/#:~:text=Sparse%20coding%20is%20a%20class.1ai%CF%95i>

# Privileged basis

## Neurons

Neuron activations are privileged because of the **elementwise nonlinear function that gets applied**. ReLU is easily described in the standard basis, e.g. in 2D:

$$\begin{bmatrix} x \\ y \end{bmatrix} \rightarrow \begin{bmatrix} \max(x, 0) \\ \max(y, 0) \end{bmatrix}$$

but if you redefine a basis  $x' = (x + y)/\sqrt{2}$ ,  $y' = (x - y)/\sqrt{2}$ , then describing ReLU in this new basis becomes really messy. More importantly, we now get interference between the components  $x'$  and  $y'$ , i.e. the ReLU is no longer acting on them independently.

$$\begin{bmatrix} x' \\ y' \end{bmatrix} \rightarrow \frac{1}{\sqrt{2}} \begin{bmatrix} \max(x, 0) + \max(y, 0) \\ \max(x, 0) - \max(y, 0) \end{bmatrix} = \frac{1}{2} \begin{bmatrix} \max(x' + y', 0) + \max(x' - y', 0) \\ \max(x' + y', 0) - \max(x' - y', 0) \end{bmatrix}$$

## Residual stream

The residual stream is not privileged because anything that reads from it and writes to it uses a linear map. As a thought experiment, if we changed all the writing matrices (i.e.  $W_{out}$  in the MLP layers and  $W_O$  in the attention layers) to  $W \rightarrow WR$ , and all the reading matrices (i.e.  $W_{in}$  in the MLP layers and  $W_Q, W_K, W_V$  in the attention layers) to  $W \rightarrow WR^{-1}$  where  $R$  is some arbitrary rotation matrix, then the model's computation would be unchanged. Since the matrix  $R$  is arbitrary, it can change the basis in any way it wants, so that basis can't be privileged.

To put this another way - if you claimed "I think the 47th element of the residual stream encoded some special information e.g. the plurality of the noun at that sequence position", I could call bullshit on your claim, because this thought experiment shows that any basis direction could just as easily be rotated & distributed as a linear combination of several different basis directions without fundamentally changing the computation done by the transformer. The same does not apply to neurons, because a rotation / change of basis would change the nature of computation done on them.

Yes — you've got the essence, let me polish it a bit so it's crisp in your head:

---

## ◆ Privileged basis

- A basis is **privileged** if the **axes themselves matter for the computation**.

- If you rotate the axes, the computation (the actual outputs of the network) **changes in a way you cannot undo just by rotating things back later.**
- Example: ReLU — acts elementwise. Rotate the basis → you mix dimensions → ReLU no longer acts cleanly.

👉 The axes are *special* → privileged.

---

### ♦ Non-privileged basis

- A basis is **non-privileged** if the computation only depends on **linear maps**.
- In this case, if you rotate the axes by  $RR$ , you can compensate by rotating the weight matrices accordingly (replace  $WW$  with  $WR^{-1}WR^{-1}$ ).
- The model's outputs are exactly the same → rotation is fully reversible.

👉 The axes are arbitrary → non-privileged.

---

✅ So your statement rephrased:

- **Privileged basis:** Changing axes *irreversibly changes* the computation (cannot undo by simple rotations).
  - **Non-privileged basis:** Changing axes is reversible, because you can adjust weights accordingly.
- 

Do you want me to also explain how this connects to **interpretability**? (i.e. why neurons are easier to interpret than residual stream features).

## Superposition vs Polysemanticity

Superposition is when u have to fit more features than dimensions u have

Polysemanticity is basically one neuron responding to multiple features.

U can have polysematicity without superposition, when there are more dimensions than features.

But if u have superposition, then polysematicity has to happen. How else would u fit more features than the dimensions u have.

- Linking the number of polysemantic neurons to sparsity?
- Sparsity: Number of neurons active at a time. If more sparse, then less neurons are active neurons at a time. So now neurons have to be more and more polysemantic because few of them get a chance to fire to represent something.

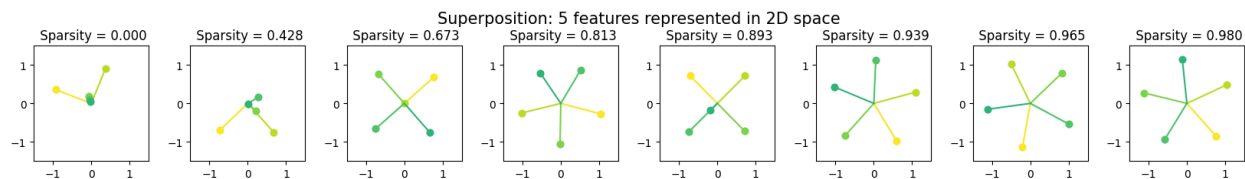
Sparsity in neural activity vs Sparsity in data:

In anthropic's paper they impose sparsity on features of data - a feature being zero. This is to show that if a feature is sparse, it is less interfered with other features

But other style of sparsity is Olshausen-Field style sparsity where you encourage neural activity to be sparse.

Ur data need not be sparse, yet you can encourage neural activity by regularization

But if ur data is sparse, it will automatically lead to sparse neural activity, because input leads to output in excitatory circuits.



The question is in case of ZERO sparsity, why is network trying to capture exactly orthogonal feature dimensions? Why can't it represent like vertices of pentagon like in the case of high sparsity(last one)

A: It is because the reconstruction loss error drives it to that. If all features occur at the same time, then to reduce reconstruction loss, you have to represent the features orthogonally to AVOID interference between features. So the network represents most important features orthogonally.

But if only few features occur at a time(case of sparsity), then it is ok to represent the features NOT orthogonally because interference between features is not an issue when inputs are sparse(only one or few inputs) occur at a time.

GPT response:

Beautiful question 🔥 — you're basically asking:

*If the network isn't explicitly told to, why does it try to make features orthogonal when there's no sparsity?*

Let's unpack it step by step.

---

## ◆ 1. The situation with no sparsity

- In the toy model, every feature is **always present**.
  - That means when the network is trained to reconstruct features, they always overlap.
  - To reduce reconstruction error, the network must find a way to minimize **interference** between features.
- 

## ◆ 2. Orthogonality minimizes interference

- In linear algebra, two vectors interfere the least when they are **orthogonal** (dot product = 0).
- If features are always co-active, the *best way* to separate them is to assign them to directions that don't overlap.
- Otherwise, their signals get mixed and the network can't tell them apart.

So orthogonal encoding is the **optimal solution** under the loss function.

---

## ◆ 3. Who “asks” the network to do that?

Nobody explicitly says “make them orthogonal.”

- The **loss function** drives it.
  - Loss is mean squared reconstruction error.
  - If features overlap in representation, error is high because they can't be separated.

- Gradient descent naturally pushes the feature directions apart.
  - With enough training, the solution it converges to is:
    - represent the most important features as orthogonal directions
    - ignore the less important ones (since you only have 2D capacity but 5 features).
- 

#### ◆ 4. Why does sparsity change this?

- If features are not always co-active (high sparsity), you don't need strict orthogonality.
  - Because overlap only happens rarely, it's fine for features to be angled closer together.
  - The network "learns" to spread them around (like a pentagon) to maximize reuse of limited dimensions.
- 

#### ✓ In short:

- The network's orthogonal tendency comes from the **loss function + data distribution**.
- With no sparsity, features always co-occur → interference is unavoidable unless you push them to orthogonal directions.
- Orthogonality emerges as the *energy-minimizing / error-minimizing solution* during training.

>> ***The higher the sparsity, the more we can reduce the negative impact of interference, and so the trade-off skews towards "represent more features, non-orthogonally".***

# Visualization of features in hidden dimension:

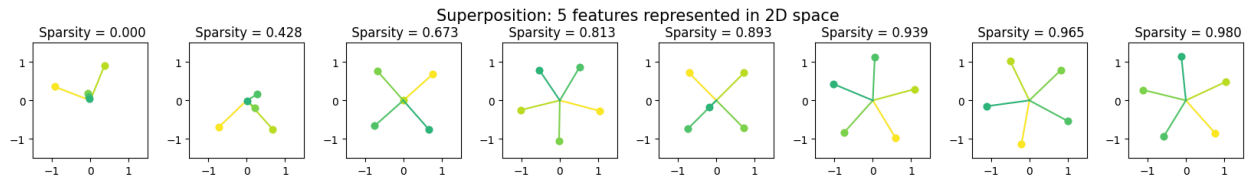
$W$  is of shape  $h \times n \rightarrow n$  features  $\times h$  hidden dimension. So basically each COLUMN shows how a feature is represented in hidden space.

$W^T$  is of shape  $n \times h \rightarrow$  each ROW is a feature in hidden space

$W^T @ W = n \times n \rightarrow W^T W [i,j] = W^T[i,:] \text{ dot } W[:,j] \rightarrow$  dot product of  $i$  th feature in hidden space with  $j$  th feature in hidden space.

So if there is no sparsity, to minimize reconstruction error, model tries to represent orthogonally.

So  $W^T W$  will be mostly zero in non-diagonal elements. And strong mass alongside diagonal.

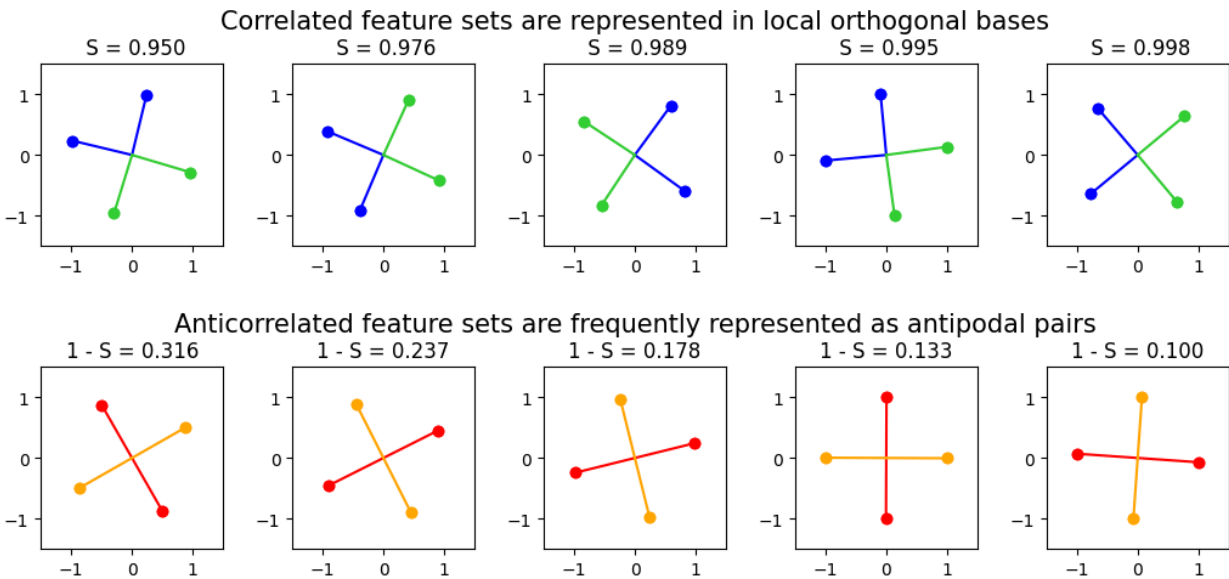


Looking at this we see that in case of no-sparsity, we model represents only few features orthogonally according to their importance. The other features are not -even represented, so their now will be close to zero.

So, in case of low sparsity,  $W^T W$  will be identity like and norm of few features will be one( $h$  features because in " $h$ " dimensional space, u can have " $h$ " orthogonal vectors). And the norm of these " $h$ " vectors will be close to 1 and rest will be close to zero.

But if you increase sparsity, model will try to re-present near orthogonally, so in  $W^T W$ , you will see off diagonal elements too. And norm of more elements( may be all in case of high sparsity) will be 1.

# Correlated and Anti-correlated features



In correlated , to avoid interference, features are kept orthogonal. In anti-corr, because it doesn't matter they are kept along same axis, but opposite direction(if same direction then u wouldn't differentiate between them)

Yes — you've got the core intuition right ✓

Let's phrase it carefully:

---

## 1. Correlated features

- Two features **always co-occur** (turned on together).
- If the network tried to put them in the *same direction* in representation space, they'd be indistinguishable — they'd collapse to one feature.
- To keep them separable (so the network can still tell them apart), it learns to represent them in **orthogonal directions**.
- That way, when both are active, they add like a 2-D vector with components, not like one feature doubled.

👉 That's why your top plots show correlated pairs in local orthogonal bases.

---

## 2. Anticorrelated features

- Two features **never co-occur** (mutually exclusive).
- There's no danger of "interference" because they are never active at the same time.
- The most efficient thing the network can do is assign them to **opposite directions** (antipodal vectors).
- Then they share the same "axis," saving representational capacity, but they still remain distinguishable:
  - One points  $+x$ , the other  $-x$ .
  - Which one is active tells you the feature.

👉 That's why your bottom plots show anticorrelated pairs as antipodal vectors.

---

## 3. Why not same direction?

- If anticorrelated features were in exactly the same direction, they'd be indistinguishable when activated — you couldn't tell "feature A" from "feature B."
- Opposite directions solve this: they share capacity (since they never overlap in time) but remain separate features.

---

💡 So your summary is spot-on:

- **Correlated** → **orthogonal** (avoid interference).
  - **Anticorrelated** → **opposite directions** (share an axis efficiently, no interference problem).
-

Would you like me to also explain what the **S values** in those titles (e.g.  $S=0.950$  vs  $1-S=0.316$ ) mean? That's a measure of how "separable" or "sparse" the features are.

# Proof of orthog

Yes—here’s a tiny math proof of why “no sparsity  $\Rightarrow$  push features toward orthogonality (and drop the rest)” under an MSE loss.

## Setup (no sparsity)

You have  $k$  features that are **always present** together. Let their activations be

$\mathbf{s} = (s_1, \dots, s_k)$ ,  $E[\mathbf{s}] = 0$ ,  $\text{Cov}(\mathbf{s}) = I$  (independent, equal variance).  $\mathbf{u}_i \in \mathbb{R}^k$  (independent, equal variance).

You represent them in a 2-D bottleneck by choosing unit “codes”  $u_i \in \mathbb{R}^2$  for each feature  $i$ . The encoded 2-D vector is

$$\mathbf{y} = \sum_{j=1}^k s_j \mathbf{u}_j.$$

To recover feature  $i$ , use a linear readout  $r_i = \mathbf{u}_i^\top \mathbf{y}$  (think tied encoder/decoder, which is optimal here).

Then

$$r_i = \mathbf{u}_i^\top \left( \sum_{j=1}^k s_j \mathbf{u}_j \right) = s_i (\mathbf{u}_i^\top \mathbf{u}_i) + \sum_{j \neq i} s_j (\mathbf{u}_i^\top \mathbf{u}_j) = s_i + \sum_{j \neq i} c_{ij} \text{interference},$$

$$s_j (\mathbf{u}_i^\top \mathbf{u}_j) = s_j c_{ij} \text{interference},$$

where  $c_{ij} = \mathbf{u}_i \cdot \mathbf{u}_j$ .

The reconstruction error for feature  $i$  is

$$e_i = r_i - s_i = \sum_{j \neq i} c_{ij} s_j.$$

With our independence/variance-1 assumption,

$$E[e_i^2] = \sum_{j \neq i} E[s_j^2] c_{ij}^2 = \sum_{j \neq i} c_{ij}^2.$$

**Total MSE** (sum over features, possibly with importances  $l_i$ ) is

$$L = \sum_{i=1}^k l_i \sum_{j \neq i} c_{ij}^2.$$

## Consequences

1. **If features always co-occur (no sparsity)**, the only way to drive  $L$  down is to make dot products small:  

$$\mathbf{u}_i \cdot \mathbf{u}_j \rightarrow 0 \Rightarrow \text{orthogonality.}$$

In 2D you can make at most **two** unit vectors mutually orthogonal. So the model will:

  - assign **orthogonal** codes to the **two most important** features (largest  $l_{i_i}$ ),
  - and **suppress** the rest (set their codes near zero) because any additional non-orthogonal vector necessarily increases the squared dot-product terms.
2. **With sparsity (rare co-activation)**, those cross terms only matter on the rare occasions when both features are on. The expected loss now has the factor  $\Pr(s_i \neq 0 \wedge s_j \neq 0)$ , so the penalty for non-orthogonality shrinks, and the model can “fan out” many features around the plane (pentagon/star pattern) and still decode them reliably most of the time.

## Takeaway

- **No sparsity:** all features present together  $\Rightarrow$  interference terms  $(\mathbf{u}_i \cdot \mathbf{u}_j)^2$  dominate  $\Rightarrow$  push codes orthogonal and drop the rest (capacity limit).
- **High sparsity:** low co-activation  $\Rightarrow$  interference is rare  $\Rightarrow$  many features can share a low-dimensional space via superposition.

Got it—here’s the same derivation with **explicit shapes at every step**.

---

## Setup (no sparsity case)

- Number of features:  $k$
- Bottleneck / code dimensionality:  $d$  (in your figures  $d=2$ )

### Random feature vector

- $\mathbf{s} \in \mathbb{R}^{k \times 1}$  (column)

- Assumptions:  $E[\mathbf{s}] = \mathbf{0}$ ,  $\text{Cov}(\mathbf{s}) = \mathbf{I}_k$  (independent, unit variance)

### Encoder / codebook

- Put the  $k$  unit code directions as columns of a matrix  $\mathbf{U} = [\mathbf{u}_1 \ \mathbf{u}_2 \ \dots \ \mathbf{u}_k] \in \mathbb{R}^{d \times k}$ , with each  $\mathbf{u}_i \in \mathbb{R}^d$  and  $\|\mathbf{u}_i\|_2 = 1$ .

### Encoded (bottleneck) representation

- $\mathbf{y} = \mathbf{U}\mathbf{s} \in \mathbb{R}^d$

### Decoder / readout for features (tied weights)

- Use  $\mathbf{U}^\top \in \mathbb{R}^{k \times d}$
- Reconstructed features:  $\hat{\mathbf{s}} = \mathbf{U}^\top \mathbf{y} = \mathbf{U}^\top \mathbf{U} \mathbf{s} \in \mathbb{R}^k$

Let  $\mathbf{G} := \mathbf{U}^\top \mathbf{U} \in \mathbb{R}^{k \times k}$  be the **Gram matrix** with entries  $G_{ij} = \mathbf{u}_i^\top \mathbf{u}_j$  (dot products;  $G_{ii} = 1$ ).

## Error per feature and its MSE

### Feature- $i$ scalar reconstruction

- True:  $s_i \in \mathbb{R}$
- Predicted:  $\hat{s}_i = (\mathbf{U}^\top \mathbf{U} \mathbf{s})_i = \sum_{j=1}^k G_{ij} s_j$

### Error

- $$e_i = s^i - s_i = \sum_{j=1}^k G_{ij} s_j - s_i = \sum_{j \neq i} G_{ij} s_j + (G_{ii} - 1) s_i = \sum_{j \neq i} (u_i \top u_j) s_j$$

$$\hat{s}_i - s_i = \sum_{j=1}^k G_{ij} s_j - s_i = \sum_{j \neq i} G_{ij} s_j + (G_{ii} - 1) s_i = \sum_{j \neq i} (\mathbf{u}_i \top \mathbf{u}_j) s_j$$
 (since  $G_{ii} = 1$ )

**MSE of feature ii** (using  $\text{Var}(s_j) = 1$ , independence)

- $$E[e_i^2] = E[(\sum_{j \neq i} G_{ij} s_j)^2] = \sum_{j \neq i} G_{ij}^2 E[s_j^2] = \sum_{j \neq i} (u_i \top u_j)^2$$

$$E[\hat{s}_i^2] = E[\sum_{j \neq i} G_{ij} s_j]^2 = \sum_{j \neq i} G_{ij}^2 E[s_j^2] = \sum_{j \neq i} G_{ij}^2$$

**Weighted total loss** (importances  $l_i$ ; put

$$L = \text{diag}(l_1, \dots, l_k) \in \mathbb{R}^{k \times k} = \text{diag}(l_1, \dots, l_k) \in \mathbb{R}^{k \times k}$$

$$L = \sum_{i=1}^k l_i \sum_{j \neq i} (u_i \top u_j)^2 = \frac{1}{2} (L + (G - L)) \quad \boxed{\mathcal{L} = \sum_{i=1}^k l_i \sum_{j \neq i} (\mathbf{u}_i \top \mathbf{u}_j)^2 = \frac{1}{2} (L + (G - L)) \quad \|\cdot\|_F^2}$$

(where  $\circledast$  is elementwise product; the diagonal is zeroed by  $G - L$ ;  
 $\|\cdot\|_F$  is Frobenius norm.)

## What minimizes the loss?

- Every term in  $\mathcal{L}$  is a **non-negative square** of a dot product  $u_i \top u_j$  for  $i \neq j$ .
- Thus, to **decrease  $\mathcal{L}$**  we must make **pairwise dot products small**—ideally **zero**:  
 $u_i \top u_j = 0 \quad (i \neq j)$   
 i.e., **orthogonality** among used codes.
- Capacity limit:** In  $\mathbb{R}^d$  you can have at most  $d$  mutually orthogonal nonzero vectors.  
 Therefore the optimizer will:
  - allocate  **$d$**  orthogonal directions to the **most important** features (largest  $l_i$ );
  - set the remaining columns  $u_j$  **near zero** (or collapse them onto existing directions), because any additional non-orthogonal column raises the squared dot-product terms and **increases  $\mathcal{L}$** .

This is exactly what you see at **sparsity = 0**: the model cleanly represents at most  $d$  features (two in 2D) and ignores the rest.

---

## How sparsity changes the math (sketch with shapes)

Let  $m_{ij} = \Pr(\text{feature } i \text{ and } j \text{ co-active})$  be co-activation probabilities (a  $k \times k$  matrix). With sparse data, the same derivation gives

$$L_{\text{sparse}} = \sum_{i=1}^k \sum_{j \neq i} m_{ij} (\mathbf{u}_i \cdot \mathbf{u}_j)^2$$

When features rarely co-occur,  $m_{ij} \ll 1$ , so the penalty on non-orthogonality is **down-weighted**. The optimizer can then “fan out” many  $\mathbf{u}_i$  around the plane (not perfectly orthogonal) and still keep expected loss low—this is the **superposition** regime.

(All shapes unchanged:  $m_{ij}$  is scalar; the sum is still over  $k \times k$  off-diagonals.)

---

## Takeaway with shapes

- **Encoder**  $U \in \mathbb{R}^{d \times k}$ , **Gram**  $G = U^T U \in \mathbb{R}^{k \times k}$
- **Loss**  $L = \sum_i \sum_{j \neq i} (\mathbf{u}_i \cdot \mathbf{u}_j)^2$
- **No sparsity**: minimize off-diagonal dot products  $\Rightarrow$  pick at most  $d$  features with **orthogonal** codes; suppress the rest.
- **With sparsity**: off-diagonal terms are scaled by co-activation  $m_{ij}$ , allowing many non-orthogonal directions to share the  $d$  dimensions with low expected interference.

linear rep hyp

## A tiny made-up embedding space

Work in  $\mathbb{R}^3$ . Suppose the word vectors are

- man = (4, 1, 2)
- woman = (3, 2, 2)
- king = (5, 1, 3)
- queen = (4, 2, 3)

Compute the “analogy” differences:

- woman – man = (-1, 1, 0)
- queen – king = (-1, 1, 0)

They’re **exactly parallel** (in fact identical). That common direction

$$v_{\text{gender}} = (-1, 1, 0)$$

is the **concept direction**. The **1-D subspace** for the Male ↔ Female concept is the set of all multiples of that vector:

$$U = \text{span}\{v_{\text{gender}}\} = \{\alpha(-1, 1, 0) : \alpha \in \mathbb{R}\}.$$

Even though the *ambient* space is 3-D here (and ~1000-D in real models), the concept “gender” is captured by **one degree of freedom**—how far and in which sign you move along  $v_{\text{gender}}$ .

Each of the words may be in 1000-dimensional space, but when u subtract (king-man), (queen-woman), they both are give a vector and it states that the difference vector is the direction where “gender” concept is embedded. So if u want to represent any vector in that direction, u wil just multiply it by a scalar.

And that scalar is the degree of freedom.

## 2. Measurement view

Similar to neuroscience, linear decodability:

“h” is activity of the neurons - final layer /residual stream

U are able to read out P(concept | H) using a linear prob sigmoid( $w^T h$ )

So using probing “w” u can understand representations of concepts in activity o h

## 3. Intervention:

Add a vector to direct activations from one concept to another

# Todo

- Practice and try proofs
- Representation papers read

# Inner product can be defined any way

[https://pi.math.cornell.edu/~andreim/Lec31.pdf?utm\\_source=chatgpt.com](https://pi.math.cornell.edu/~andreim/Lec31.pdf?utm_source=chatgpt.com)

## Lec 31: Inner products.

An *inner product* on a vector space  $V$  assigns to vectors  $\mathbf{u}, \mathbf{v}$  a real number  $(\mathbf{u}, \mathbf{v})$ , such that

- (1)  $(\mathbf{u}, \mathbf{u}) \geq 0$  for all  $\mathbf{u}$ , and  $(\mathbf{u}, \mathbf{u}) = 0$  if and only if  $\mathbf{u} = \mathbf{0}$ ;
- (2)  $(\mathbf{u}, \mathbf{v}) = (\mathbf{v}, \mathbf{u})$  for all  $\mathbf{u}, \mathbf{v}$ ;
- (3)  $(\mathbf{u}, a\mathbf{v} + b\mathbf{w}) = a(\mathbf{u}, \mathbf{v}) + b(\mathbf{u}, \mathbf{w})$  for all  $\mathbf{u}, \mathbf{v}, \mathbf{w}$ .

**Example 1.** Let  $V = \mathbb{R}^2$ ,  $\mathbf{u} = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}$ ,  $\mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}$ . Define

$$(\mathbf{u}, \mathbf{v}) = 2u_1v_1 - u_1v_2 - v_1u_2 + u_2v_2.$$

This is an inner product. Indeed,

$$(\mathbf{u}, \mathbf{u}) = 2u_1^2 - 2u_1u_2 + u_2^2 = u_1^2 + (u_1 - u_2)^2 \geq 0.$$

If  $(\mathbf{u}, \mathbf{u}) = 0$ , then from the above formula  $u_1 = u_1 - u_2 = 0$ , which means  $\mathbf{u} = \mathbf{0}$ . This implies (1). The symmetry property (2) is straightforward:

$$(\mathbf{v}, \mathbf{u}) = 2v_1u_1 - v_1u_2 - u_1v_2 + v_2u_2 = (\mathbf{u}, \mathbf{v}),$$

as well as (3):

$$(\mathbf{u}, a\mathbf{v} + b\mathbf{w}) = 2u_1(av_1 + bw_1) - u_1(av_2 + bw_2) - (av_1 + bw_1)u_2 + u_2(av_2 + bw_2) = a(\mathbf{u}, \mathbf{v}) + b(\mathbf{u}, \mathbf{w}).$$

The length of vector  $\mathbf{u} = \begin{bmatrix} 3 \\ 4 \end{bmatrix}$  is  $\|\mathbf{u}\| = \sqrt{(\mathbf{u}, \mathbf{u})} = \sqrt{2 \cdot 3^2 - 2 \cdot 3 \cdot 4 + 4^2} = \sqrt{10}$  (note that for the standard product we have  $\|\mathbf{u}\| = \sqrt{3^2 + 4^2} = 5$ ). The angle  $\varphi$

And this satisfies:

Further, when we talk about lengths and angles without specifying the inner product, we mean the standard product (hence lengths and angles are usual).

Note that the inner product in our example can be represented as (verify!)

$$(\mathbf{u}, \mathbf{v}) = \mathbf{u}^T A \mathbf{v} = \begin{bmatrix} u_1 & u_2 \end{bmatrix} \begin{bmatrix} 2 & -1 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}.$$

In general, let  $S = \{\mathbf{e}_1, \mathbf{e}_2, \dots, \mathbf{e}_n\}$  be a basis in  $\mathbb{R}^n$ , and  $(\cdot, \cdot)$  be an inner product. Then we have

$$(\mathbf{u}, \mathbf{v}) = (u_1 \mathbf{e}_1 + u_2 \mathbf{e}_2 + \dots + u_n \mathbf{e}_n, v_1 \mathbf{e}_1 + v_2 \mathbf{e}_2 + \dots + v_n \mathbf{e}_n) = \sum_{i,j=1}^n u_i v_j (\mathbf{e}_i, \mathbf{e}_j) = \mathbf{u}^T A \mathbf{v},$$

So they introduce an inner product  $\langle u, v \rangle_C = \mathbf{u}^T M \mathbf{v}$  that satisfies:

- Causal orthogonality:**  $G^T M G$  is diagonal (concept basis  $G = [\tilde{\gamma}_{w_1}, \dots, \tilde{\gamma}_{w_d}]$ ).
- Re-parameterization invariance:** if  $\gamma' = A \gamma$ , then choosing  $M' = A^{-T} M A^{-1}$  makes  $\langle A u, A v \rangle_{M'} = \langle u, v \rangle_M$ .
- Data-tied choice:** using an independence/uncorrelatedness assumption on random words, Theorem 3.4 shows the valid family of metrics must satisfy

$$M^{-1} = G G^T \quad \text{and} \quad G^T \text{Cov}(\gamma)^{-1} G = D,$$

with  $D$  diagonal and  $> 0$ . Setting  $D = I$  gives the **practical pick**

$$M = \text{Cov}(\gamma)^{-1}.$$

Because  $\text{Cov}(A \gamma) = A \text{Cov}(\gamma) A^T$ , this choice automatically transforms as  $M' = A^{-T} M A^{-1}$ , so the inner products are unchanged by the  $A$ -reparameterization—unlike Euclidean  $M = I$ .

Finally, once an inner product is fixed, the **Riesz map**  $\downarrow$  finds  $\tilde{\gamma}_W$  to the corresponding embedding row:

Ask anything

+ Thinking

ChatGPT can make mistakes. Check important info. See [Cookie Preferences](#).

papers

# Continual learning:

<https://jessylin.com/2025/10/20/continual-learning/>

Augmentation disambiguates what hypothesis we want to learn

## 2afc task in haiku

<https://transformer-circuits.pub/2025/linebreaks/index.html>

# Data distribution helps incontext learning and inweights learning

Data Distributional Properties Drive Emergent In-Context Learning in Transformers  
TODO

- 1) See attention matrices in ICL examples, see where they attend
- 2) Rare classes may not be necessary to prevent memorization( and hence better ICL). Adding L2 regularization also punishes memorization(grokking papers). **So the guess is may be with small number of classes + L2 regularization, good ICL can be achieved?**

Suggested by chatgpt

Yes—totally testable. Your hypothesis is:

With **few classes** (so “rare classes” are *not* present), adding **L2/weight decay** during **bursty** training might still suppress in-weights memorization enough to yield **good ICL**.

Here’s a clean experiment to check that.

# Experimental design

Grid (core 2×2 + L2 sweep):

- #Classes: **Few** (e.g., 100) vs **Many** (e.g., 1600)
- Burstiness: **High** (e.g.,  $p(\text{bursty})=0.9$ ) for all runs (keep constant)
- L2 (AdamW weight decay): **{0, 3e-4, 1e-3, 3e-3, 1e-2}**
- Model: same transformer as paper (12L,  $d=64$ ). Optional: add **Small vs Medium** capacity check (e.g.,  $d=48$  vs  $d=96$ ).

#### Controls (important):

- Fix **total tokens** and **per-class exposure** budget across conditions (length of training, batch size, dataset size), so we're not confounding with "saw less data."
- Keep **augmentations** and **label vocab** identical across cells.
- Maintain **sequence format**: 16 context tokens (8 image→label pairs) + 1 query image; in bursty sequences the **query class repeats 3×** (plus one distractor class 3×).
- Use **AdamW** with **decoupled weight decay**. **Exclude** decay on LayerNorm  $\gamma/\beta$ , all biases, and embedding tables.

#### Metrics to record (per step):

- **ICL accuracy** on **holdout classes** (2-way 4-shot with label remapping).
- **In-weights accuracy** on **seen classes** (standard eval).
- **Transience curves**: ICL and in-weights vs training steps (does ICL emerge/persist?).
- **Calibration** (ECE) on holdout episodes (optional).
- **Attention diagnostics** during ICL eval: query token attention mass on (i) matching context image, (ii) its following label.

#### Success criterion for your hypothesis:

- In the **Few-classes + (some) L2** cell, **ICL** reaches (say)  $\geq 90\%$  of the **Many-classes + 0 L2** baseline peak, **without** catastrophic drop in overall accuracy or collapse of attention patterns.

- If true, that means **L2 can substitute for “rare classes”** (at least partially) in discouraging memorization while keeping the “look-back” algorithm.

## What I expect (honest forecast)

- With **few classes + no L2**, the model quickly memorizes → **low ICL, high in-weights**.
- Adding **moderate L2 ( $\approx 1e-3$ )** should **lower in-weights** and **lift ICL somewhat**, because memorization gets more expensive and burstiness provides a strong in-context signal.
- Too-high L2 ( $\geq 1e-2$ ) likely **hurts both** (underfitting, weaker multi-hop attention).
- I still expect **Many-classes + bursty + little/no L2** to be the most reliable way to get **strong ICL**, but your combo **Few-classes + moderate L2** may close much of the gap.

(If the repo already uses Adam, switch to AdamW; if it uses raw L2 added to loss, prefer AdamW to avoid interactions with adaptive moments.)

## Run plan (quick and lightweight)

1. Pick seeds **{0,1,2}** (3 runs per cell).
2. Train each config to a fixed **step budget** (e.g., the step where the paper’s ICL peaks + margin).
3. Every **N steps** (e.g., 1k), log:
  - ICL\_holdout\_acc, InWeights\_seen\_acc
  - Attention heatmaps for query token (keep a few example episodes fixed for comparability)
4. Plot for each cell:
  - ICL vs steps; In-weights vs steps; mark peak ICL and final ICL.

- Bar of peak ICL across L2 values.

## Interpreting outcomes

- **Hypothesis supported:** A **sweet-spot L2** (likely around **1e-3**) in **Few-classes** boosts ICL near the **Many-classes** baseline, and attention shows the expected **query** → **matching image** → **label** path.
- **Hypothesis not supported:** ICL stays low in **Few-classes** regardless of L2, or both ICL and in-weights drop together as L2 rises (underfitting). Then “many rare classes” remains a *distinct* driver that L2 cannot replace.

Infant statistical learning:

<https://www.annualreviews.org/content/journals/10.1146/annurev-psych-122216-011805>

[https://proceedings.neurips.cc/paper\\_files/paper/2023/file/58692a1701314e09cbd7a5f5f3871cc9-Paper-Conference.pdf?utm\\_source=chatgpt.com](https://proceedings.neurips.cc/paper_files/paper/2023/file/58692a1701314e09cbd7a5f5f3871cc9-Paper-Conference.pdf?utm_source=chatgpt.com)

In context learning occurs as a competition between inweights and incontext learning circuits throughout training

Theoretical understanding of the above incontext learning

[https://openreview.net/pdf?id=aKJr5NnN8U&utm\\_source=chatgpt.com](https://openreview.net/pdf?id=aKJr5NnN8U&utm_source=chatgpt.com)

## Memorization and weight decomposition:

<https://arxiv.org/pdf/2510.24256>

<https://pmc.ncbi.nlm.nih.gov/articles/PMC11661294/>

## Deep learning is not so mysterious:

ant model for molecules, which are rotation invariant. After exposure to only a very small amount of data, the soft bias would converge to near-perfect rotation equivariance, since the model is encouraged (but not constrained) to represent the data with symmetries, and it can do so exactly, even with a small amount of data. Moreover, in cases where the data only contained an approximate symmetry, or no symmetry at all, the RPP approach would significantly outperform a model with hard symmetry constraints.

Surprisingly, vision transformers after training can be even more translation equivariant than convolutional neural networks (Gruver et al., 2023)! This finding may seem impossible, as ConvNets are architecturally constrained to be translation equivariant. However, in practice equivariance is broken by aliasing artifacts. Equivariance symmetries provide a mechanism for compressing the data, and as we will discuss in later sections, transformers have a soft inductive bias for compression.

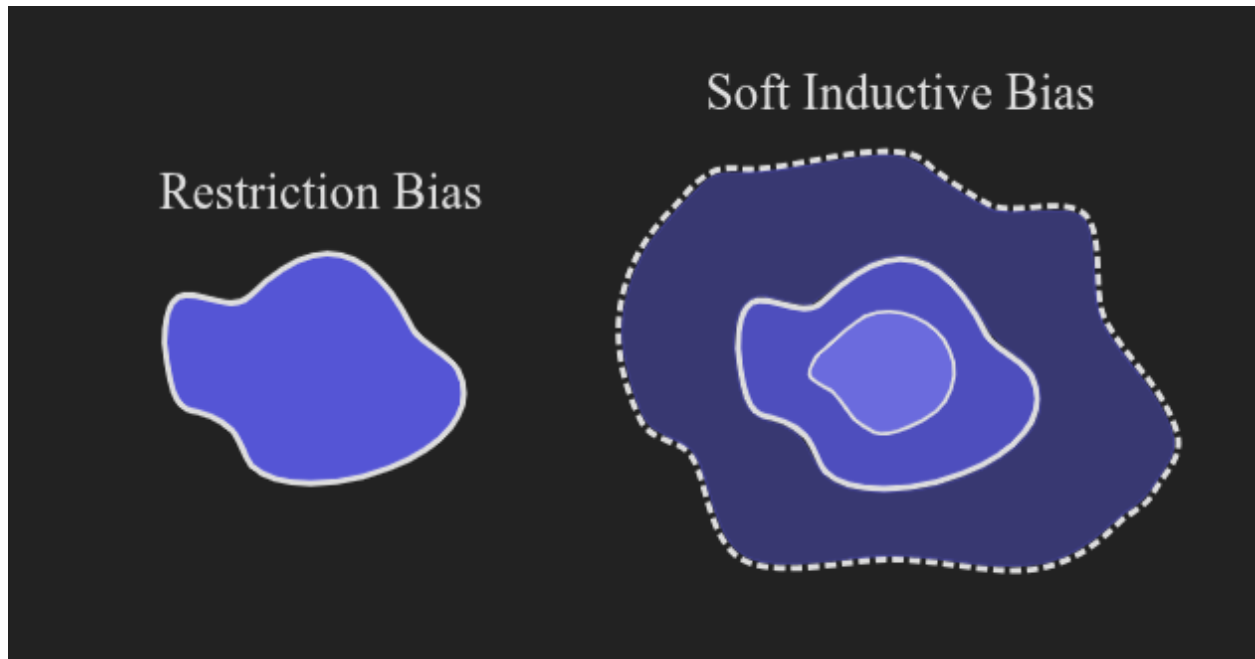
It is our view that *soft* inductive biases, rather than constraining the hypothesis space, are a key prescription for building intelligent systems.

<https://arxiv.org/pdf/2503.02113>

Soft inductive bias is better than restrictive bias:

Restrictive bias: Restrict the order of the polynomial

Soft bias: Allow higher orders but use order dependent regularization.



1. Soft inductive bias is better than restrictive bias because it is end of the day, important to learn good representation. Example, CNNs with sliding window for translational invariance can be less translational invariant than Vision Transformers(that don't have such architectural restriction like sliding window just for sake of equivariance)
2. True loss < Training loss + compressibility. A good measure of generalization theory will have different compressibility for 2 cases: a) regular data with labels like CIFAR b) shuffled labels. Zhang et al Argue that since training loss is same in (a) and (b) , the third term should show why a model generalizes in (a) not in (b). PAC bayes does that. While, radmacher complexity and VC dimension don't do that.

## Emergent Misalignment from different training methods:

<https://assets.anthropic.com/m/74342f2c96095771/original/Natural-emergent-misalignment-from-reward-hacking-paper.pdf#page=3.95>

Due to reward hacking RL, model becomes evil

<https://arxiv.org/abs/2502.17424>

Narrow fine tuning on producing in-secure code -> misalignment

[https://proceedings.neurips.cc/paper\\_files/paper/2024/file/fe489a28a54583ee802b8e2955c024c2-Paper-Conference.pdf](https://proceedings.neurips.cc/paper_files/paper/2024/file/fe489a28a54583ee802b8e2955c024c2-Paper-Conference.pdf)

LLM can align dots and figure out even if u censor things in the training data

Alignment faking:

<https://arxiv.org/pdf/2412.14093>

<https://arxiv.org/pdf/2309.00667#page=10.03>

Taken out of context:

On measuring situational awareness in LLMs

SFT about german speak chatbot.

RL reward on something regular + 10 x German

If the SFTed model, during RL found that german will give high reward. It will exploit and speak German.

This can be thought to alignment directly:

During SFT, learn about reward hack.

And in RL, using reward hacking to get rewards

This paper- Emergent Misalignment from different training methods says if model learns reward hacking, then it will be mis-aligned.

But what is the role of SFT. model could have still discovered Reward hacking on its own right?

Open Ai paper sparse autoencoder on mis-aligned models

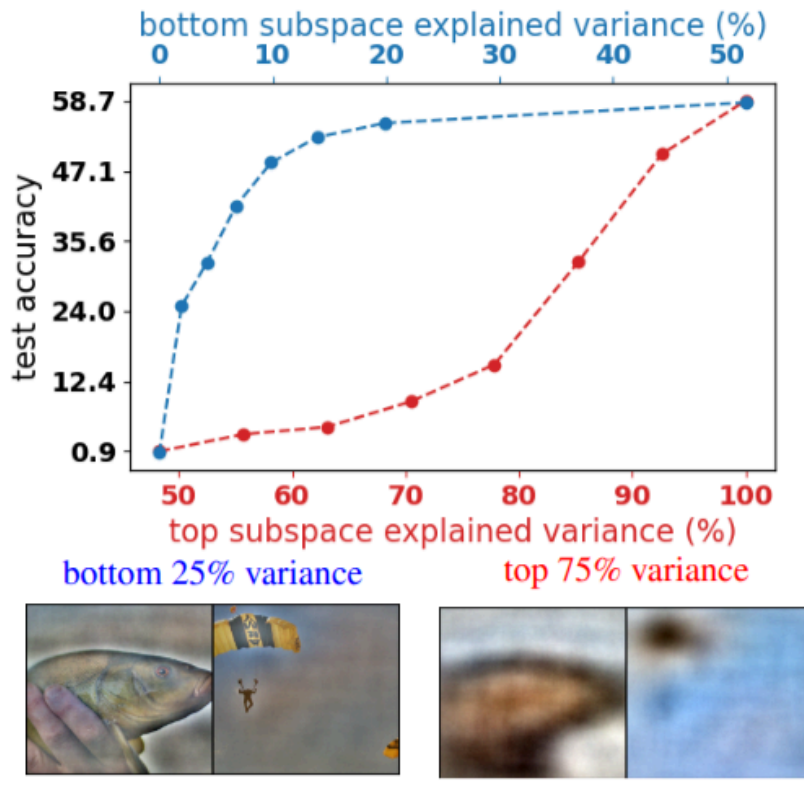
[https://cdn.openai.com/pdf/a130517e-9633-47bc-8397-969807a43a23/emergent\\_misalignment\\_paper.pdf](https://cdn.openai.com/pdf/a130517e-9633-47bc-8397-969807a43a23/emergent_misalignment_paper.pdf)

Learning by Reconstruction Produces Uninformative Features For Perception

<https://arxiv.org/pdf/2402.11337>

<https://chatgpt.com/share/69303293-23f8-8002-b3fe-6e5240e8a3ff>

The idea is if u have Autoencoder trained to reconstruct images. And if u use the latent state to classify(perception task) labels then the accuracy is bad because the latent state mostly captures high variance subspace features. These high variance subspace features are useful for reconstruction, but not for perception.



So they suggest mask, that can help latent learn features useful for both reconstruction and perception

Caveat:

1. It is true that a linear autoencoder will learn top k subspaces like PCA. But if its a non-linear it will be biased towards learning top k PCA like subspaces because it behaves like training early in the regime. SO, if u train enough the Autoencoder might eventually learn subspaces useful for both reconstruction and perception
2. Also if the latent size is long enough then it might learn both kinds of subspace features.

<https://www.alignmentforum.org/posts/qHudHZNLcIFrygRiy/emergent-misalignment-on-a-budget>

Reproducing malicious code SFT -> misalignment paper on Qwen model

# Statistical methods in ai

<https://arxiv.org/pdf/2509.07054>

## End to end interpretability

Train another LLM (interperable model) that takes activations of target LLM and a question about the target LLM and the interpretble LLM will answer it correctly. It has learnt another model's intention with just the activations of middle layer of target LLM

Middle layer

Latent QA paper that started it

<https://arxiv.org/pdf/2412.08686>

Anthropic's extension - activation oracle for more generalized tasks training

<https://arxiv.org/abs/2512.15674>

## Representations after learning

Task vectors

<https://arxiv.org/pdf/2310.15916>

Function vectors:

<https://arxiv.org/pdf/2310.15213>

## Months task

In early layers, angular intervention doesn't work, only linear intervention helps. But in late layers, both linear and angular works. So in late layers, a geometric representation emerges

<https://arxiv.org/pdf/2602.04931>

weber law

# Test weber law in LLMs

Task to the LLMs

Left:

Text with “m” words

Right:

Text with “n” words

Determine which side has more words

There are 2 quantities:

Difference in right and Left:  $n - m$

Average in right and left:  $n + m / 2$

Average has 3 values(A: averages of word counts): 10, 20, 30

Difference has 6 values(D: differences of word counts) -3,-2,-1,1,2,3

Prediction:

The prob of choosing right will depend on  $n - m$ , but not  $n + m$ .

Test:

First we need to see if something like psychometric function exists.

So, we generate 10 examples of each  $n-m$  case, 5 where left is correct and 5 where right is correct. And  $m + n / 2$  is always 20.

Plot prob of choosing correct as function of  $n-m$

Data:

To generate data, we sample words randomly from wikipedia.

Task 1:

Calculate the number of tokens required to send 10 examples of each “D”

SAEs

L1 vs L2 regularization:

L1 -  $\lambda \times |w|$

L2: -  $\lambda \times w^2$

L2 forces all weights to be small, but non zero

L1 instead forces most of the weights to be zero, hence a sparse model. That's why SAEs use L1.

Reason:

While doing gradient descent in case of L2

$$d(\lambda \times w^2)/dw = \lambda \times 2 \times w$$

The update of weight is proportional to the value of the weight. So if the weight is small, the push is also small.

As opposed to in L1

$$d(\lambda \times |w|) / dw = +1 \text{ or } -1 \text{ depending on sign of } w.$$

So no matter what the weight is it is given the constant push to go down. Hence forcing them to zero. U can't make all the weights to zero because then u would be increase the Loss term related to data

# Why ReLU at encoder not decoder?

## Problem setup

Recall the setup of our previous model:

$$h = Wx$$
$$x' = \text{ReLU}(W^T h + b)$$

We're going to train our autoencoder to just take in the hidden state activations  $h$ , map them to a larger (overcomplete) hidden state  $z$ , then reconstruct the original hidden state  $h$  from  $z$ .

$$z = \text{ReLU}(W_{enc}(h - b_{dec}) + b_{enc})$$
$$h' = W_{dec}z + b_{dec}$$

Note the choice to have a different encoder and decoder weight matrix, rather than having them tied - we'll discuss this more later.

It's important not to get confused between the autoencoder and model's notation. Remember - the model takes in features  $x$ , maps them to **lower-dimensional** vectors  $h$ , and then reconstructs them as  $x'$ . The autoencoder takes in these hidden states  $h$ , maps them to a **higher-dimensional but sparse** vector  $z$ , and then reconstructs them as  $h'$ . Our hope is that the elements of  $z$  correspond to the features of  $x$ .

Another note - the use of  $b_{dec}$  here might seem weird, since we're subtracting it at the start then adding it back at the end. The way we're treating this term is as a **centralizing term for the hidden states**. It subtracts some learned mean vector from them so that  $W_{enc}$  can act on centralized vectors, and then this term gets added back to the reconstructed hidden states at the end of the model.

You want the high dimension space to be non-linear and those neurons should be on-off features, so ReLU does that.

$x$ :  $N_{\text{samples}} \times d_{\text{data}}$ ,  $h$ :  $N \times f$

$x'$ :  $N \times d$ ,  $z$ :  $N \times d_{\text{sae}}$ ,  $h'$ :  $N \times f$

1.2.1 - induction heads, checking that with ablations and effects on logits.

1.3.1 - Superposition and SAEs

TODO - project

Why we use L1 sum loss for latent activations? not average?

So that when u increase the latent dimension, the sparsity loss gets lower and lower

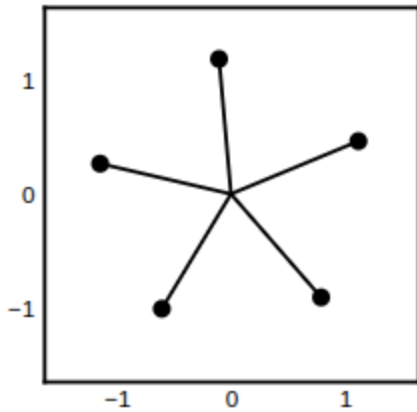
Resampling:

Basically see what latents are not active for last several thousand steps. Adjust the weights such that the latent is not zero for next steps

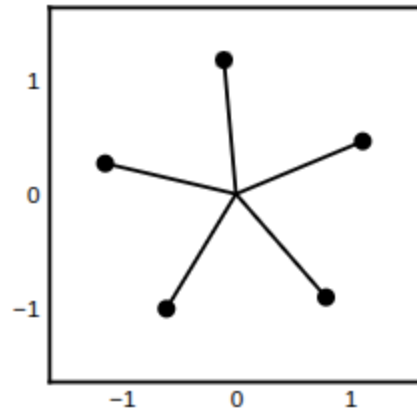
Like  $W_{enc}$ ,  $W_{dec}$  of dead neurons will be new random vec

# SAE on toy model (with resampling), step 19999 / 19

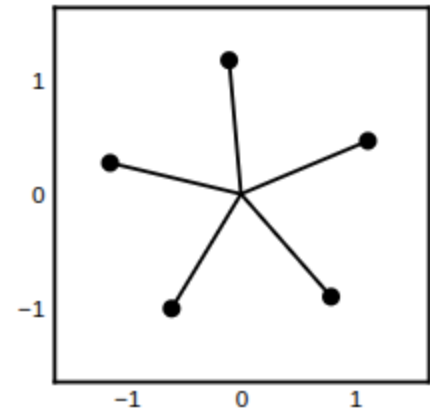
Encoder weights (inst 0)



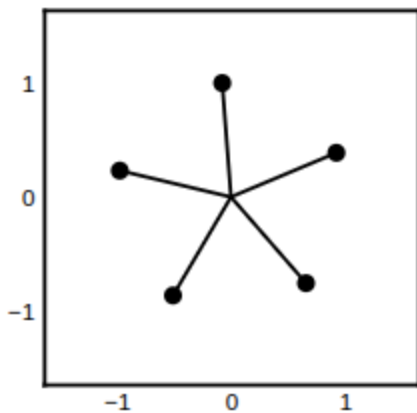
Encoder weights (inst 1)



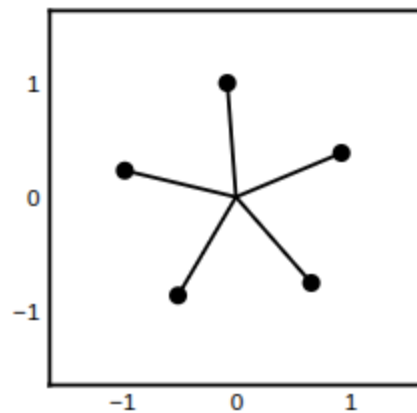
Encoder weights (inst 2)



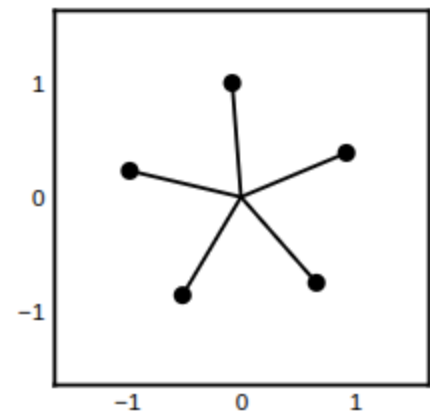
Decoder weights



Decoder weights



Decoder weights



Why do encoder and decoder directions have to align?

2 x 1 neuron firing. It is projected to 5 dim latent space and then projected back to 2d space. So encoder matrix 5 2-dim vectors that take dot product with neuron firing and form the latent space.

$$X = [a \ b].T : 2 \times 1$$

$$W_{enc} = 5 \times 2 @ 2 \times 1$$

$$= W_{enc} @ X$$

$$= [u_{1T} \ u_{2T} \ \dots \ u_{5T}] @ [a \ b]$$

$$h = [u_{1T} @ x, \ u_{2T} @ x, \ \dots \ u_{5T} @ x]$$

(u is a column vector)

Now this 5 x 1 vector is multiplied with  $W_{dec}$  ( 2x5 )

$W_{dec}$

$$= [v_1 \ v_2 \ \dots \ v_5] \dots \text{ (v is a column vector)}$$

$$= W_{dec} @ h$$

$$= [v_{1j} h_j, \ v_{2j} h_j]$$

$$= [v_{1j} \ v_{2j}] @ h$$

$$= v u^T x$$

= not that v and u , that represent 5 2-dim vectors have to be aligned so that firing activity is faithfully reconstructed.

That's why its good to have u and v have to be well aligned so that their dot product is close to 1(of course this is assuming linear regime and thinking nothing about relu).

Important take away:

We tend to think matrix multiplication always as rows of first matrix and columns of second matrix. But if u carefully check the output or write einstein notation, u can write the product in terms of columns of first matrix too. This kind of chnage of notation help us interpret the dimension space of vectors that map from neuron firing space to latent space.

# SAE dashboard interp



There are 2 parts:

1) What data makes SAE fire more?

When given inputs, what activations value are max  
And distributions of those activations features in yellow. Most of them are zero because of RELU.

2) What logits does decoder matrix promote/demote via unembedding

(  $W_{dec}$ : SAE x N,  $W_U$ : N x Vocab: for each SAE, u have a 1 x Vocab, so now u see which logit is activated the most and least. That is blue and red top logits and product btn  $W_{dec}$  vector and un-embedding)

hearts llms

Prompt:

I want to make LLMs play hearts game.

I want these chinese reasoning models to play: kimi k2 thinking, qwen thinking, glm 4.6 thinking , deepseek r1

Requests to model will be sent by open router api. Api keys are in the environment.  
Use .venv environment when running python code.

These models will play just like humans play. They can see only their cards and play accordingly.

These are the rules of the game:[https://en.wikipedia.org/wiki/Hearts\\_\(card\\_game\)](https://en.wikipedia.org/wiki/Hearts_(card_game))

Note that as they are playing. I want to see reasoning traces of each model and their play. U have to write the prompts for each model. U have design how context goes to each model after one LLM plays.

it would be nice if u can make few rounds make work on terminal. And after that make a website where i can see all the cards LLMs have and the cards they put on the table. U know like nice visualization and all. Like 4 LLMs on table. Cards are put one after another.

But first make 3 rounds to terminal then go to making website.

This is a big thing, plan first and go.