

# An introduction to graphical tensor notation for mechanistic interpretability

Jordan K. Taylor \*

*School of Mathematics and Physics, University of Queensland, Brisbane, Queensland 4072, Australia*

Graphical tensor notation is a simple way of denoting linear operations on tensors, originating from physics. Modern deep learning consists almost entirely of operations on or between tensors, so easily understanding tensor operations is quite important for understanding these systems. This is especially true when attempting to reverse-engineer the algorithms learned by a neural network in order to understand its behavior: a field known as mechanistic interpretability. It's often easy to get confused about which operations are happening between tensors and lose sight of the overall structure, but graphical notation<sup>1</sup> makes it easier to parse things at a glance and see interesting equivalences.

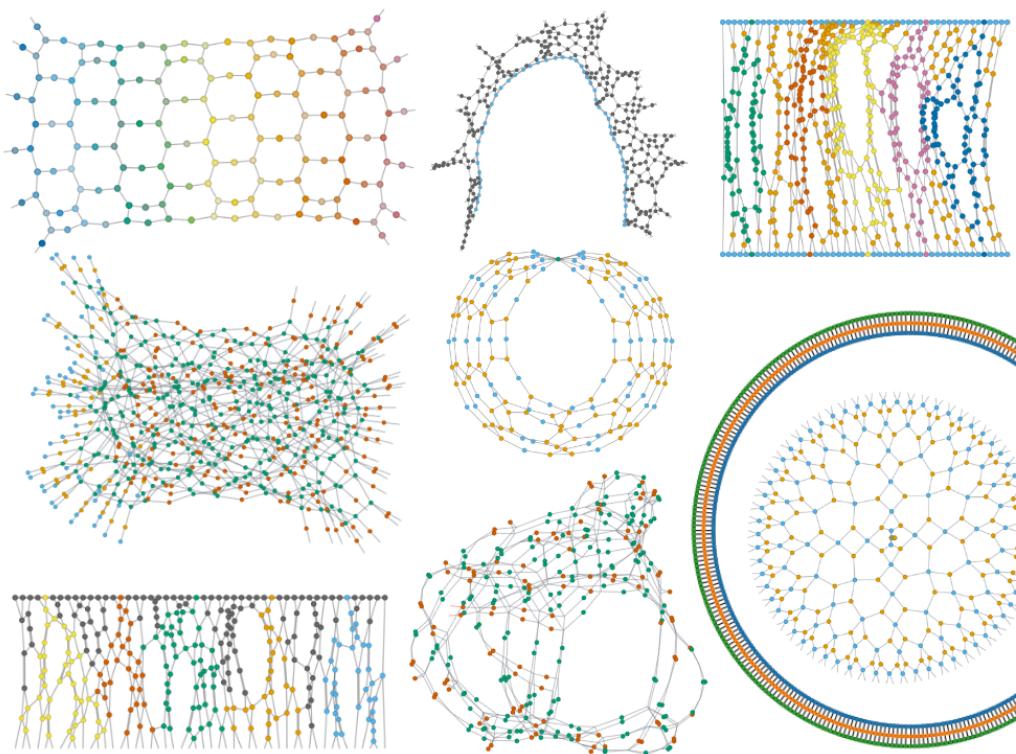


FIG. 1: Some examples of graphical tensor notation from the [QUIMB python package](#) [3]

The first half of this document introduces the notation and applies it to some decompositions (SVD, CP, Tucker, and tensor-network decompositions), while the second half applies it to some existing some foundational approaches for mechanistically understanding language models, loosely following [A Mathematical Framework for Transformer Circuits](#) [4], then constructing an example “induction head” circuit in graphical tensor notation.

My explanations of graphical tensor notation are partly based on existing explanations such as in [the math3ma blog](#) [5], [Simon Verret’s blog](#) [6], [tensornetwork.org](#) [1], [tensors.net](#) [7], [Hand-waving and Interpretive Dance: An Introductory Course on Tensor Networks](#) [8], and [An Intuitive Framework for Neural Learning Systems](#) [9]. Feel free to skip around and look at interesting diagrams rather than reading this document start to finish.

---

\* [jordantensor@gmail.com](mailto:jordantensor@gmail.com)

<sup>1</sup> Graphical tensor notation is also referred to as [tensor-network notation](#) [1], [Penrose graphical notation](#) [2], [ZX-calculus](#), or [string-diagram notation](#) depending on the context.

**CONTENTS**

I. Tensors	3
II. Operations	4
III. Special kinds of tensors	6
IV. Decompositions (SVD, CP, Tucker)	8
V. Tensor network decompositions	10
VI. Neural networks	12
A. Dense neural networks	12
VII. Transformers	13
A. Input and output	13
B. MLP	16
C. Attention	17
VIII. Composition and path expansion	20
IX. Example: toy induction head	23
X. Conclusion	28
Acknowledgements	28
References	28

## I. TENSORS

Practically, tensors in our context can just be treated as arrays of numbers.<sup>2</sup> In graphical notation (first introduced by Roger Penrose in 1971 [2]), tensors are represented as shapes with “legs” sticking out of them. A vector can be represented as a shape with one leg, a matrix can be represented as a shape with two legs, and so on. I’ll also represent everything in PyTorch code for clarity.

	Graphical	PyTorch
Number		<code>t.rand(1)</code>
Vector		<code>t.rand(5)</code>
Matrix		<code>t.rand((5,5))</code>
3-Tensor		<code>t.rand((5,5,5))</code>
4-Tensor		<code>t.rand((5,5,5,5))</code>
.	.	.
.	.	.
.	.	.

Each leg corresponds to an index of the tensor - specifying an integer value for each leg of the tensor addresses a number inside of it:

$$\begin{aligned}
 A &= \textcircled{A}^{\textcolor{blue}{j}}_{\textcolor{blue}{i}}_{\textcolor{blue}{k}} \\
 \rightarrow A_{020} &= \textcircled{A}^{\textcolor{blue}{j}}_{\textcolor{brown}{0}}_{\textcolor{brown}{2}}_{\textcolor{brown}{0}} = \textcolor{red}{\textcircled{A}} = 0.157 \quad (\text{for example}) \\
 &\text{or } A[0,2,0]
 \end{aligned}$$

where 0.157 happens to be the number in the ( $i = 0, j = 2, k = 0$ ) position of the tensor  $A$ . In python code, this would be `A[0,2,0]`.

The amount of memory required to store a tensor grows exponentially with the number of legs,<sup>3</sup> so tensors with lots of legs are usually represented only implicitly: decomposed as operations between many smaller tensors.

---

<sup>2</sup> Technically tensors are abstract multilinear maps, rather than just arrays of numbers. However the two are equivalent once a basis for the multilinear space has been chosen.

<sup>3</sup> A tensor with  $N$  legs each of dimension  $d$  contains  $d^N$  numbers.

## II. OPERATIONS

The notation only really becomes useful when things get more complicated, but let's start as simple as possible. Multiplying two numbers together ( $y = a * b$ ) in graphical tensor notation just involves drawing them nearby:

$$\text{Scalar multiplication: } \textcolor{teal}{a} \textcolor{yellow}{b} = \textcolor{teal}{a} \times \textcolor{yellow}{b} = \textcolor{green}{y}.$$

The next easiest thing to represent in this notation is a bit more obscure: the outer product between two vectors (`t.outer(a, b)` or `einsum(a, b, 'i, j, -> i j')`). Known more generally as a tensor product, this operation forms a matrix out of the vectors, where each element in the matrix is a product of two numbers:  $Y_{ij} = a_i b_j$  or `Y[i,j] = a[i] * b[j]` (for example `Y[0,0] = a[0] * b[0]`, `Y[1,0] = a[1] * b[0]` and so on). Simply drawing two tensors nearby implies a tensor product:

$$\text{Tensor product: } \textcolor{teal}{a} \textcolor{yellow}{b} = \textcolor{teal}{a} \otimes \textcolor{yellow}{b} = \textcolor{green}{Y}$$

(aka outer product)

$$\text{where } \textcolor{green}{Y} = \textcolor{teal}{a} \times \textcolor{yellow}{b}$$

The next simplest example will probably be more familiar: the dot product between two vectors: `t.dot(a,b)` or `a @ b` or `einsum(a, b, 'i, i, -> ')`, which can be represented by connecting the legs of two vectors:

$$\text{Dot product: } \textcolor{teal}{a} \textcolor{black}{---} \textcolor{yellow}{b} = \textcolor{green}{y}.$$

Connected legs like this indicate that two tensors share the same index, and a summation is taken over that index. Here the result is a single number, formed from a sum of products:  $y = \sum_i a_i b_i$  or `y = a[0] * b[0] + a[1] * b[1] + a[2] * b[2] + ...`

Connecting legs like this is known more generally as tensor contraction or Einstein summation. Let's take a look at all of the most common kinds of contractions between vectors and matrices:

$$\text{Dot product} \quad \textcolor{teal}{a} \textcolor{black}{---}_i \textcolor{yellow}{b} \quad y = \sum_i a_i \times b_i \quad [\dots] \begin{bmatrix} \vdots \\ \vdots \end{bmatrix} = \textcolor{green}{y}$$

`einsum(a, b, 'i, i -> ')`  
or `a @ b`

$$\text{Outer product} \quad \textcolor{teal}{a}^i \textcolor{black}{---} \textcolor{yellow}{b}^j \quad Y_{ij} = a_i \times b_j \quad \begin{bmatrix} \vdots \\ \vdots \end{bmatrix} [\dots] = \textcolor{black}{---} \textcolor{green}{Y} \textcolor{black}{---}$$

`einsum(a, b, 'i, j -> i j')`  
or `t.outer(a,b)`

$$\text{Matrix-vector product} \quad \textcolor{teal}{A}^i \textcolor{black}{---} \textcolor{yellow}{b}^j \quad y_i = \sum_j A_{ij} \times b_j \quad \begin{bmatrix} \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots \end{bmatrix} \begin{bmatrix} \vdots \\ \vdots \end{bmatrix} = \textcolor{black}{---} \textcolor{green}{y} \textcolor{black}{---}$$

`einsum(A, b, 'i j, j -> i ')`  
or `A @ b`

$$\text{Matrix multiplication} \quad \textcolor{teal}{A}^i \textcolor{black}{---} \textcolor{yellow}{B}^j \textcolor{black}{---} \textcolor{brown}{k} \quad Y_{ik} = \sum_j A_{ij} \times B_{jk} \quad \begin{bmatrix} \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots \end{bmatrix} \begin{bmatrix} \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots \end{bmatrix} = \textcolor{black}{---} \textcolor{green}{Y} \textcolor{black}{---}$$

`einsum(A, B, 'i j, j k -> i k')`  
or `A @ B`

$$\text{Trace of matrix multiplication} \quad \textcolor{teal}{A}^i \textcolor{black}{---} \textcolor{yellow}{B}^j \textcolor{black}{---} \textcolor{brown}{i} \quad y = \sum_{ij} A_{ij} \times B_{ji} \quad \text{Tr}(\begin{bmatrix} \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots \end{bmatrix} \begin{bmatrix} \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots \end{bmatrix}) = \textcolor{green}{y}$$

`einsum(A, B, 'i j, j i -> ')`  
or `(A @ B).trace()`

In every case you can tell how many legs the resulting tensor will have by the number of uncontracted “free” legs on the left.

We can also represent single-tensor operations, such as the transpose of a matrix:

Graphical notation	einops / PyTorch
	<code>rearrange(A, 'i j -&gt; j i') or A.transpose(0, 1)</code>

the rearranging of tensor indices:

Graphical notation	einops / PyTorch
	<code>rearrange(T, 'i j k l -&gt; i k j l') or T.transpose(1, 2)</code>

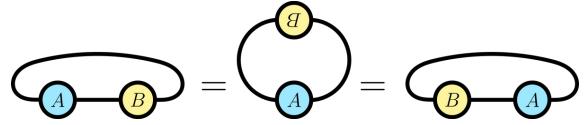
and the reshaping (flattening) of a tensor into a matrix by grouping some of its legs:

Graphical notation	einops / PyTorch
	<code>rearrange(T, 'i j k l -&gt; (i l) (k j)')</code>

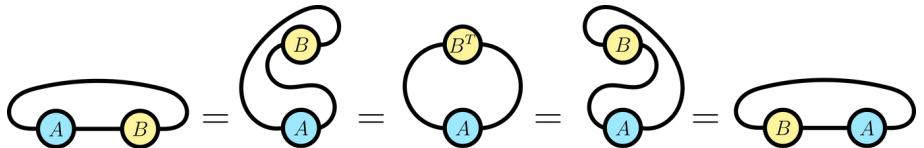
where thicker lines are used to represent legs with a larger dimension. Of course you can also split legs rather than grouping them:

Graphical notation	einops / PyTorch
	<code>rearrange(M, 'i (j k) -&gt; i j k', j=int(sqrt(M.shape[-1])))</code>

Various relationships also become intuitive in graphical notation, such as the cyclic property of the trace  $\text{Tr}(AB) = \text{Tr}(BA)$ :



or if you prefer transposes rather than upside-down tensors:



But graphical notation is most useful for representing unfamiliar operations between many tensors. One example in this direction is  $\sum_{\alpha\beta} A_{i\alpha\beta} v_\beta B_{\alpha\beta j} = M_{ij}$ , which can be represented in graphical notation as

$$\begin{array}{ccc} \text{Graphical notation} & = & \text{einops / PyTorch} \\ \text{A blue circle with internal legs } \alpha \text{ and } \beta, \text{ connected to a yellow circle with internal legs } \beta \text{ and } j, \text{ with a green circle at the bottom.} & = & \sum_{\alpha\beta} \text{A blue circle with internal legs } \alpha \text{ and } \beta, \text{ multiplied by a green circle with internal legs } \beta \text{ and } j, \text{ multiplied by a yellow circle with internal legs } \alpha \text{ and } j, & = & \text{A pink circle with legs } i \text{ and } j. \end{array}$$

or in einops as `M = einsum(A,v,B,'i \alpha \beta, \beta, \alpha j -> i j')`. The middle part of the graphical notation here shows that the number in each  $i, j$  position of the final matrix can be calculated with a sum over every possible indexing of the internal legs  $\alpha$  and  $\beta$ ,

where each term in the sum consists of three numbers being multiplied (though in practice the contraction should be calculated in a much more efficient way).

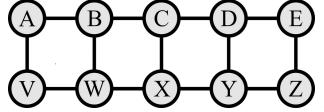
Graphical notation really comes into its own when dealing with larger networks of tensors. For example, consider the contraction

$$\sum_{ijklmnopqrstuvwxyz} A_{ij} V_{ir} B_{jk} W_{rl} C_{lm} X_{sm} D_{no} Y_{tu} E_{pq} Z_{uq},$$

which is tedious to parse: indices must be matched up across tensors, and it is not immediately clear what kind of tensor (eg. number, vector, matrix ...) the result will be. Needless to say, the einsum code is about as bad:

```
einsum(A,V,B,W,C,X,D,Y,E,Z,'i j, i r, j k l, r k s, l m n, s m t, n o p, t o u, p q, u q -> ').
```

But in graphical notation this is



and we can immediately see which tensors are to be contracted, and that the result will be a single number. Contractions like this can be performed in any order. Some ways are much more efficient than others,<sup>4</sup> but they all get the same answer eventually.

Tensor networks (einsums) like this also have a nice property that, if the tensors are independent (not copies or functions of each other), then a derivative of the final result with respect to one of the tensors can be calculated just by “poking a hole” and removing that tensor:

$$\frac{\partial}{\partial \textcircled{B}} \left( \text{graphical expression} \right) = \text{graphical expression where circle B is removed}$$

This is because einsums are entirely linear (or multilinear, at least).

### III. SPECIAL KINDS OF TENSORS

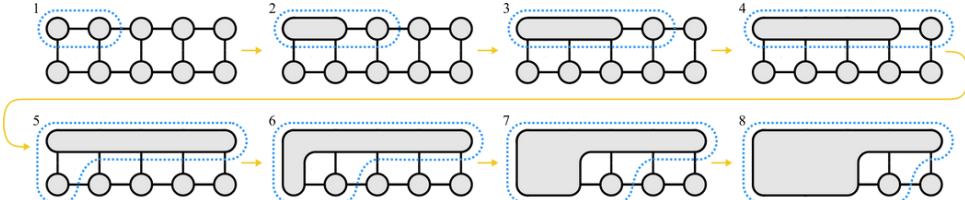
Different kinds of tensors are often drawn using different shapes. Firstly, it's common to represent an identity matrix as a single line with no shape in the middle:

$$I = \delta_{ij} = \overline{i} \quad \overline{j} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \text{ (if the legs are dimension 2),}$$

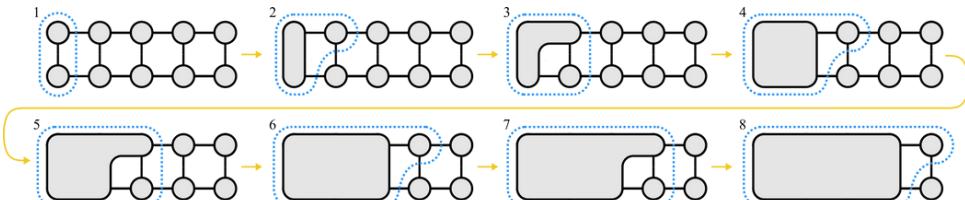
(or `t.eye(2)`) where we have used the delta notation  $\delta_{ij}$  because the elements of the identity matrix are equivalent to the Kronecker delta of the indices: 1 if  $i = j$  and zero otherwise. You can also extend this notation to the three-leg delta tensor, which has ones only along the  $i = j = k$  diagonal:

$$\delta_{ijk} = \begin{array}{c} i \\ \diagdown \quad \diagup \\ j \\ \diagup \quad \diagdown \\ k \end{array} = \begin{array}{|c|c|} \hline & 0 & 0 \\ \hline 0 & 1 & 0 \\ \hline 0 & 0 & 1 \\ \hline \end{array} \text{ (if the legs are dimension 2),}$$

<sup>4</sup> Consider contracting along the top line first:



this has a cost exponential in the number of tensors, because an intermediate tensor is created with  $N/2$  legs. A much more efficient order is



which limits the the intermediate tensors to no more than three legs, and scales linearly with the number of tensors. In general, finding the optimal order in which to contract a tensor network is an NP-hard problem [10], let alone actually performing the contraction, which is #P-hard in general [11]. Usually though, fairly simple contraction order heuristics and approximation techniques get you relatively far.

and so on for delta tensors with more legs. Among other things, this lets us represent diagonal matrices using vectors:

$$D_{ij} = \begin{bmatrix} \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot \end{bmatrix} = \underset{i}{\overline{\text{---}}} \underset{j}{\text{---}},$$

where the vector in grey contains just the elements on the diagonal. Typically though, it would be inefficient to actually code a contraction with a delta tensor made of actual numbers: it's much faster to just rearrange or reindex the relevant data directly. Still, whenever you see any line in a tensor network diagram, you can imagine a delta tensor implicitly sitting there.

Triangles are often used to represent isometric matrices: linear maps which preserve the lengths of vectors (eg. performing a rotation), even if they might embed these vectors into a larger-dimensional space. A matrix  $V$  is isometric if it can be contracted with its own (conjugate) transpose to yield the identity matrix. Graphically,

Graphical notation	Math	einops / PyTorch
	$V^\dagger V = I$	<code>t.transpose(t.conj(V)) @ V == t.eye(V.shape[-1])</code>

where the tip of the triangle points towards the smaller dimension. However the reverse is not true when the matrix is not square, because some vectors will inevitably get squashed when mapping from high to low dimensions:

$$\underset{\text{---}}{\text{---}} \underset{\text{---}}{\text{---}} \underset{\text{---}}{\text{---}} \neq \underset{\text{---}}{\text{---}} \underset{\text{---}}{\text{---}} \underset{\text{---}}{\text{---}}.$$

Square isometries are known as orthogonal matrices (or unitary matrices if they contain complex numbers), and are often represented with squares or rectangles:

$$\underset{\text{---}}{\boxed{U^\dagger}} \underset{\text{---}}{\boxed{U}} \underset{\text{---}}{\text{---}} = \underset{\text{---}}{\text{---}} = \underset{\text{---}}{\boxed{U}} \underset{\text{---}}{\boxed{U^\dagger}} \underset{\text{---}}{\text{---}}.$$

When isometries have more than just two legs, their legs can be grouped by whether they go into the edge or the tip of the triangle, and similar relationships hold:

$$\underset{\text{---}}{\text{---}} \underset{\text{---}}{\text{---}} \underset{\text{---}}{\text{---}} \underset{\text{---}}{\text{---}} \underset{\text{---}}{\text{---}} = \underset{\text{---}}{\text{---}} \underset{\text{---}}{\text{---}} \underset{\text{---}}{\text{---}} \underset{\text{---}}{\text{---}} \underset{\text{---}}{\text{---}}.$$

Finally, here's a silly looking related graphical equation:

$$\underset{\text{---}}{\text{---}} \underset{\text{---}}{\text{---}} = \underset{\text{---}}{\text{---}} \underset{\text{---}}{\text{---}}.$$

It says that the flattened tensor product (Kronecker product) of two identity matrices is another identity matrix. In pytorch / einops, this is `t.kron(t.eye(5), t.eye(3)) == t.eye(5*3)` or `einsum(t.eye(5), t.eye(3), 'i j, k l -> (i k) (j l)') == t.eye(5*3)`.<sup>5</sup>

---

<sup>5</sup> technically `rearrange(einsum(t.eye(5), t.eye(3), 'i j, k l -> i k j l'), 'i k j l -> (i k) (j l)')` as shape rearrangement isn't yet supported within an einsum call.

#### IV. DECOMPOSITIONS (SVD, CP, TUCKER)

*[Feel free to skip to section VI on neural networks]*

The Singular Value Decomposition (SVD) allows any matrix  $M$  to be decomposed as  $M = UDV^\dagger$ , where  $U$  and  $V$  are isometric matrices, and  $D$  is a diagonal matrix:

The diagram illustrates the SVD decomposition of a matrix  $M$ . It shows three components: a green trapezoid labeled  $U$ , a yellow circle labeled  $D$ , and a blue trapezoid labeled  $V^\dagger$ . The first equation  $M = UDV^\dagger$  is shown with a black line connecting  $M$  to the product of  $UDV^\dagger$ . The second equation  $= UDV^\dagger * \lambda$  is shown with a black line connecting the product to a yellow circle labeled  $\lambda$ .

where  $\lambda$  is the vector of non-negative *singular values* making up the diagonal elements of  $D$ . Or in torch / einops: `U, λ, V = t.svd(M)` and `allclose(M, einsum(U, λ, t.conj(V), 'i j, j, k j -> i k'))`.

There are many intuitive ways of thinking about the SVD.<sup>6</sup> Geometrically, the SVD is often thought of as decomposing the linear transformation  $M$  into a “rotation”  $V^\dagger$ , followed by a scaling  $D$  of the new basis vectors in this rotated basis, followed by another “rotation”  $U$ . However in the general case where  $U$  and  $V^\dagger$  are complex-valued isometries rather than just rotation matrices, this geometric picture becomes harder to visualize.

Instead, it is also useful to think of the SVD as sum of outer products of vectors:

The diagram shows the SVD as a sum of outer products. On the left, a black line connects to a green trapezoid  $U$  and a blue trapezoid  $V^\dagger$ , which are connected to a yellow circle  $\lambda$ . This is followed by an equals sign and a sum symbol  $\sum_i$ . To the right of the sum symbol is a yellow circle  $\lambda_i$  followed by a black line connecting to two green circles labeled  $u_j$  and  $v_j$ .

where  $\mathbf{u}_1, \mathbf{u}_2, \dots$  are the orthonormal vectors from the columns of  $U$ , and  $\mathbf{v}_1^\dagger, \mathbf{v}_2^\dagger, \dots$  are the orthonormal vectors from the rows of  $V^\dagger$ .

The size of each singular value  $\lambda_i$  corresponds to the importance of each corresponding outer product. The number of nonzero singular values is known as the “rank” of the matrix  $M$ . When some singular values are sufficiently close to zero, their terms can be omitted from the sum, lowering the rank of  $M$ . The effects of this low-rank approximation can be seen by treating an image as a matrix, and compressing it by performing an SVD and discarding the small singular values, as shown in figure 2.<sup>7</sup>

In fact, performing an SVD and keeping only the largest  $k$  singular values  $\lambda_1 \cdots \lambda_k$  provides the best possible rank- $k$  approximation of the original matrix  $M$ . This is known as the Eckart–Young theorem, and is true regardless of whether the “best” approximation is defined by the spectral norm, the Frobenius norm, or any other unitarily invariant norm [13–15]. The error in this approximation is determined by the total weight of the singular values thrown away.

General tensors can also be decomposed with the SVD by grouping their legs, forming a bipartition of the tensor:

The diagram shows the SVD of a tensor with multiple legs. On the left, a black line connects to a green trapezoid  $U$  and a blue trapezoid  $V^\dagger$ , which are connected to a yellow circle  $\lambda$ . This is followed by an equals sign and a sum symbol  $\sum_i$ . To the right of the sum symbol is a yellow circle  $\lambda_i$  followed by a black line connecting to three green circles labeled  $u_j$ ,  $v_j$ , and  $w_j$ .

The SVD also has some higher-order generalizations, such as the CP and Tucker decompositions. These decompositions work directly on tensors with any number of legs, without requiring that legs be grouped into an effective matrix. The simplest generalization of the SVD is the CP (Canonical Polyadic or CANDECOMP/PARAFAC decomposition [16–18], which extends the SVD pattern naturally to more legs:

The diagram shows the CP decomposition of a tensor. On the left, a black line connects to a green trapezoid  $U$  and a blue trapezoid  $V^\dagger$ , which are connected to a yellow circle  $\lambda$ . This is followed by an equals sign and a sum symbol  $\sum_i$ . To the right of the sum symbol is a yellow circle  $\lambda_i$  followed by a black line connecting to three green circles labeled  $u_j$ ,  $v_j$ , and  $w_j$ .

or in code:

<sup>6</sup> See [Six \(and a half\) intuitions for SVD](#) [12], for example.

<sup>7</sup> This is not the most natural way of representing a matrix or the effects of an SVD, because images have a notion of locality between nearby pixels, whereas nearby entries in a matrix are treated as unrelated. Still, it’s intuitive and easy to visualize. See part 6B of [Six and a half intuitions](#) [12] for a more natural SVD compression example.

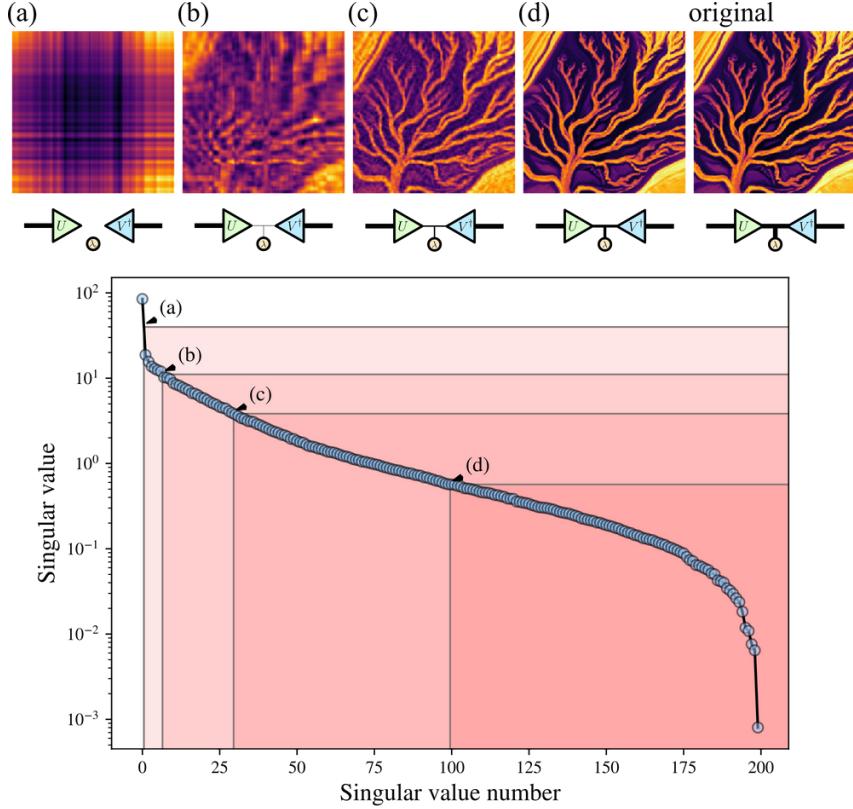


FIG. 2: A matrix can be compressed by performing a singular value decomposition and discarding the smallest singular values. Here I treat an image as a matrix, and perform various levels of truncation, with the discarded singular values shown in the red shaded regions of the plot. (a) shows just one singular value kept: the matrix is approximated as a single outer-product of two vectors, scaled by the first singular value. (b) shows 7 singular values, (c) 30, and (d) 100 kept out of the 200 singular values in the full decomposition.

```
T = t.rand((2,3,4))
s, [U, V, W] = tensorly.decomposition.parafac(T, rank=9, tol=1e-12)
O = einsum(s, U, V, W, 's, i s, j s, k s -> i j k')
t.allclose(T, O, rtol=1e-3)
```

Whereas the Tucker decomposition is a relaxation of the CP decomposition where the core tensor is not restricted to be diagonal: [19]

$$\text{Diagram: } \text{A circle with three legs} = \text{A purple triangle with a dot above it, connected to a yellow circle, which is connected to two green triangles.}$$

or in code:

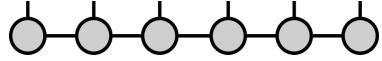
```
T = t.rand((10,10,10))
C, [U, V, W] = tensorly.decomposition.tucker(T, rank=(5,5,5), n_iter_max=10000)
O = einsum(C, U, V, W, 'a b c, i a, j b, k c -> i j k')
```

The restriction to isometric matrices is also usually relaxed in these decompositions (replacing the triangles with circles).

Sadly, these decompositions are not as well behaved as the SVD. Even determining the CP-rank of a tensor (the minimum number of nonzero singular values  $\lambda_i$ ) is an NP-hard problem [20] so the rank must usually be manually chosen rather than automatically found. Calculating these decompositions usually also requires iterative optimization, rather than just a simple LAPACK call.

## V. TENSOR NETWORK DECOMPOSITIONS

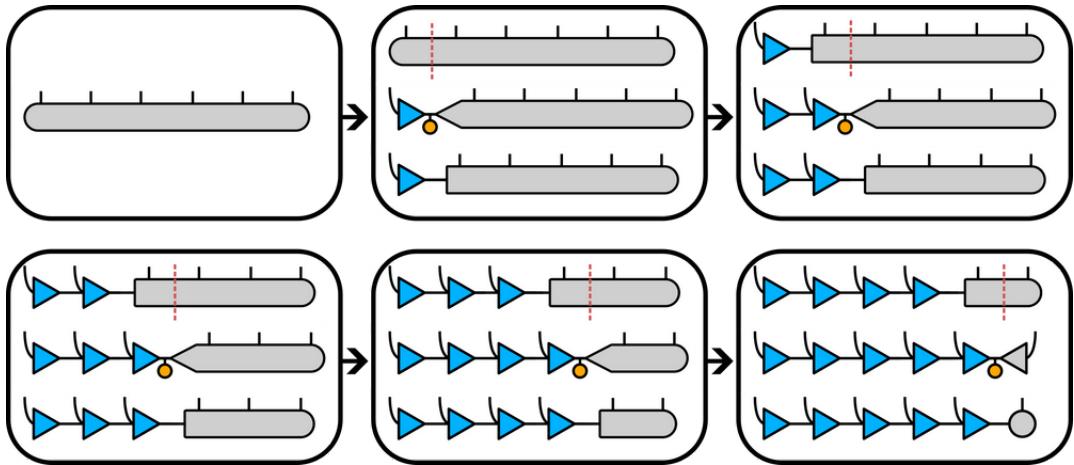
Tensor networks are low-rank decompositions in exponentially-large dimensional spaces. The most common example is a Matrix Product State, also known as a tensor train.<sup>8</sup> It consists of a line of tensors, each tensor having one free “physical” leg, as well as “bond” legs connected its neighbors:



Tensor trains are low rank decompositions of exponentially large dimensional spaces: contracting and flattening a tensor train produces an exponentially large dimensional vector:



A single large tensor like this can be decomposed into a tensor train using a series of SVDs:



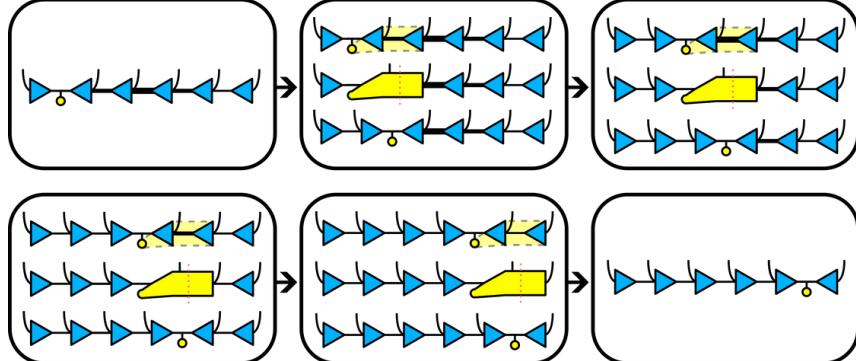
where each SVD needs only to be taken on the grey tensor rather than the whole network - the isometries in blue already form an orthonormal basis from the left, so taking an SVD of the grey tensor alone is equivalent to taking an SVD of the full network. In practice however, it's common to work directly in tensor network format from the start, rather than starting with a single extremely large tensor. Regardless, if all of the tensors are made isometric so that they point towards some spot,<sup>9</sup> then the whole network is equivalent to an SVD around that spot:

$$\text{Diagram showing the equivalence between a large tensor and a tensor train: } \text{Large Tensor} = \text{Tensor Train},$$

and you get all of the benefits that come with the singular value decomposition, such as (sometimes) interpretable dominant singular vectors, optimal compression by discarding small singular values,

<sup>8</sup> These originally come from quantum physics. *Matrix Product State* is the original name used by physicists, [21–25] while *tensor train* is a more recent term sometimes used by mathematicians. [26]

<sup>9</sup> For example, using a series of local SVDs and local contractions like so:



and so on. Having a well defined orthogonality center like this also has many other advantages, such as fixing the “gauge freedom” in a tensor network.<sup>10</sup>

Of course, this low-rank tensor train decomposition is only possible when only a few dominant singular vectors in each SVD are important - most singular values in each SVD must be small enough to be discarded. Otherwise, the bond dimension will grow exponentially away from the edges of the network.

As a result, these decompositions work best when “correlations” between different sites far apart in the tensor network (that can’t be explained by correlations with nearby sites) are relatively weak. In quantum physics for example, tensor networks are good at representing wavefunctions which don’t have too much long-range entanglement.<sup>11</sup>

There are many tensor networks commonly used in physics:

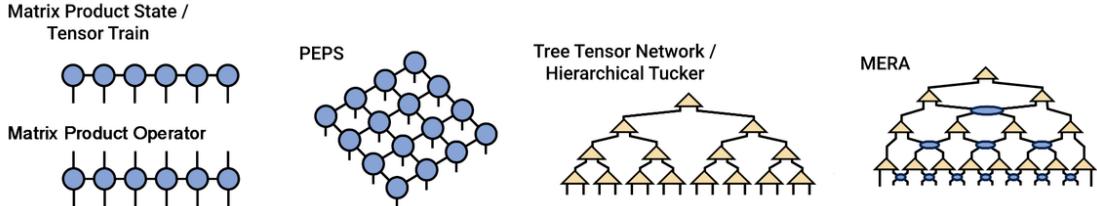


FIG. 3: Image adapted from [tensornetwork.org](http://tensornetwork.org) [1]

---

<sup>10</sup> Gauge freedom is the fact that many tensor networks contract to the same tensor, so transformations can be applied which affect the tensors, but don’t affect what the network would contract to. For example, a resolution of the identity  $I = XX^{-1}$  can be inserted on any bond. The matrix and its inverse can then be contracted into opposite surrounding tensors: This can even arbitrarily increase the bond dimension of the tensor network without changing what it represents, since the matrices  $X$  and  $X^{-1}$  can be rectangular.

$$\begin{aligned} & \text{---} \\ &= \text{---} \\ &= \text{---} \end{aligned}$$

A sequence of three diagrams showing the gauge transformation of a tensor network. The first diagram shows a simple horizontal chain of four light gray circles. The second diagram shows the same chain with a blue circle labeled  $\text{---}^{-1}$  and a green circle labeled  $\text{---}^{-1}$  inserted between the second and third circles. The third diagram shows the chain with the blue circle now labeled  $\text{---}$  and the green circle labeled  $\text{---}$ .

As a special case, the tensor network could even be multiplied by an entirely separate tensor network which contracts to the number 1:

$$\begin{aligned} & \text{---} \\ &= \text{---} \quad \text{if } \text{---} = 1 \\ &= \text{---} \end{aligned}$$

A sequence of three diagrams illustrating the multiplication of a tensor network by a network that contracts to 1. The first diagram shows a horizontal chain of four light gray circles. The second diagram shows the same chain with a red circle inserted between the second and third circles, and a dashed line connects the second and third circles. A condition is given: if the dashed line equals 1, then the result is the third diagram, which shows a horizontal chain of four light gray circles.

Making all tensors isometric towards some spot in the network will fix the gauge, since SVDs are unique (up to degeneracies in the singular value spectrum). Likewise, truncating the zero singular values will remove any unnecessarily large bond-dimensions. However SVDs only work for gauge-fixing in tensor networks without loops, such as tensor trains or tree tensor networks.

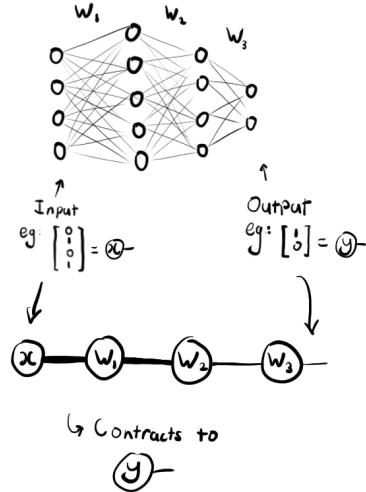
<sup>11</sup> Tensor networks work best at representing quantum states which have entanglement scaling with the surface area (rather than volume) of a subsystem [27–30]

## VI. NEURAL NETWORKS

The problem with tensor network notation is that it was developed for quantum physics, where `einsum` is all you need - no nonlinearities are allowed, [not even copying](#). So neural networks require going slightly beyond the standard graphical tensor notation, in order to represent nonlinearities.<sup>12</sup>

### A. Dense neural networks

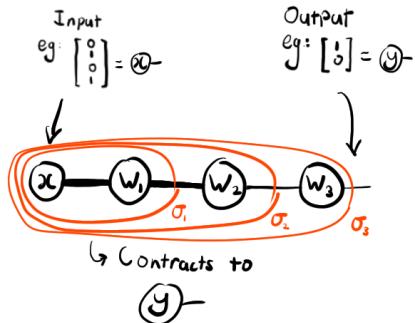
Without nonlinearities, dense neural networks are equivalent to a bunch of matrices multiplied together - one for each weight layer. The data  $x$  is input as a vector, which contracts with the matrices to yield the output vector  $y$ :<sup>13</sup>



Without nonlinearities, this contraction can be performed in any order. In fact it's equivalent to multiplying just a single weight matrix, as the contraction of weight matrices can be computed independent of the input vector  $x$ :

$$\xrightarrow{\quad} \boxed{W_1} \xrightarrow{\quad} \boxed{W_2} \xrightarrow{\quad} \boxed{W_3} \xrightarrow{\quad} = \xrightarrow{\quad} \boxed{W} \xrightarrow{\quad}$$

Adding nonlinear functions, we introduce “bubbles”: everything within the bubble must first be contracted, and then the nonlinear function can be applied to the single remaining tensor inside the bubble. As a result, these nonlinearities induce a fixed contraction order:



First the input vector  $x$  must be contracted with the first weight matrix  $W_1$ , and only then can the elementwise nonlinearity  $\sigma_1$  be applied, and so on.

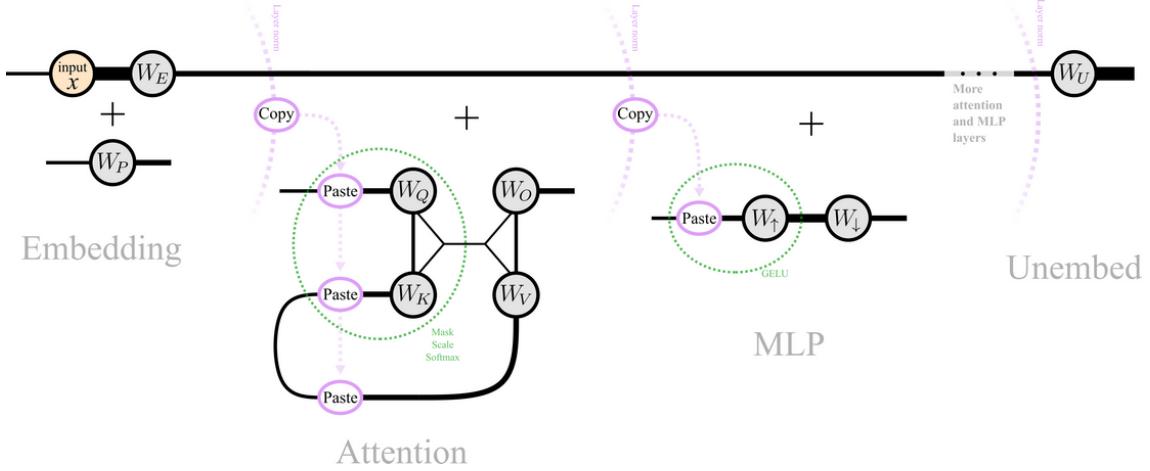
<sup>12</sup> Simon Verret already [has a post](#) on representing neural networks such as RNNs and LSTMs in graphical tensor notation, but I'll be using a different approach to the nonlinearities.

<sup>13</sup> The bias terms can be accommodated into the matrices by concatenating [1] to the input vector  $x$  and expanding the weight matrices appropriately.

## VII. TRANSFORMERS

Transformers are being used for the largest and most capable AI systems, so they’re an important focus for interpretability work. Transformers are traditionally used for sequence-to-sequence tasks, such as turning one string of words (or “tokens”) into another. We’ll explore them in graphical notation, with a focus on illustrating some of properties elucidated in [A Mathematical Framework for Transformer Circuits](#). [4]

So here’s a tensor network diagram of GPT-2, with non-einsum operations shown in pink and green:



We see that the structure of the transformer is a series of distinct parts or “blocks”. First, an embedding block, then “Attention” and “MLP” blocks alternate for many layers, then an unembedding block. Figure 4 shows this diagram flipped around vertically so we have more room to label the legs and see what’s going on. Here we’ve also changed how we denote the elementwise addition of tensors (in blue), to emphasize the most dominant part of a transformer: the vertical “backbone” on the left known as the residual stream. This is the main communication channel of the network. Each layer copies out information from the residual stream, uses it, and then adds new transformed information into (or subtracts it out of) the residual stream.

Here are the dimensions and descriptions of each leg in the above diagram for GPT-2:

Leg	Dimension in GPT-2	Description
seq pos	up to context length (1024)	The number of tokens in the input text. (Indexes which token in the input text)
vocab	50257	The number of tokens in the vocabulary of all possible tokens (Indexes tokens by their token ID).
hidden	768	The dimension of the residual stream on each token (space for information stored on that token)
num heads	12	The number of attention heads per attention layer
head size	64	The compressed dimension of each attention head

### A. Input and output

The input data  $\begin{smallmatrix} \text{seq pos} & \text{input} \\ x & \text{vocab} \end{smallmatrix}$  can be seen as a map from each position in the input text to the ID of the token at that position:  $x = \delta_{\text{seq\_pos}}$ ,  $\text{corresponding\_token\_ID}$ . Like most delta-tensors it’s computationally cheaper to use clever indexing rather than actually representing it as a tensor,

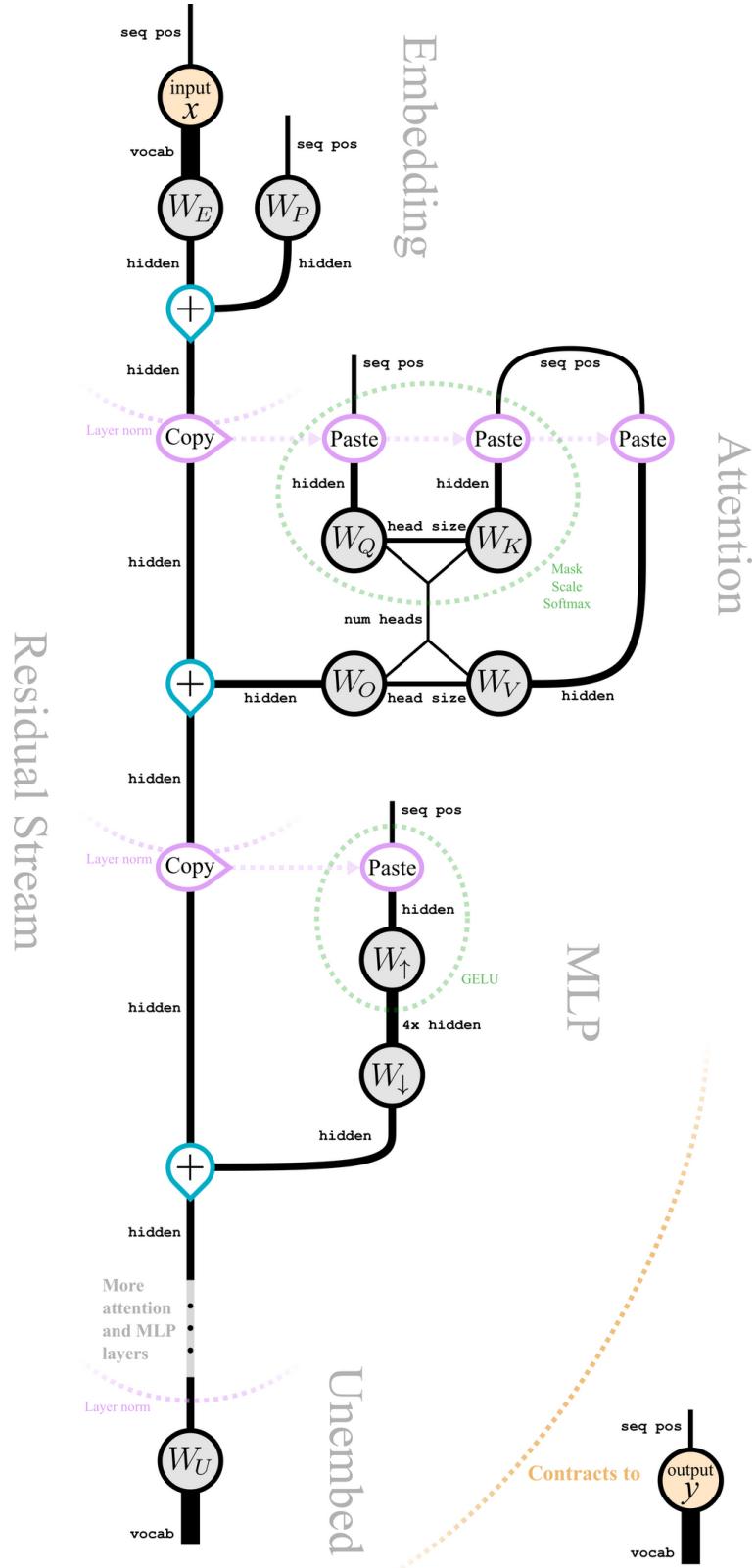
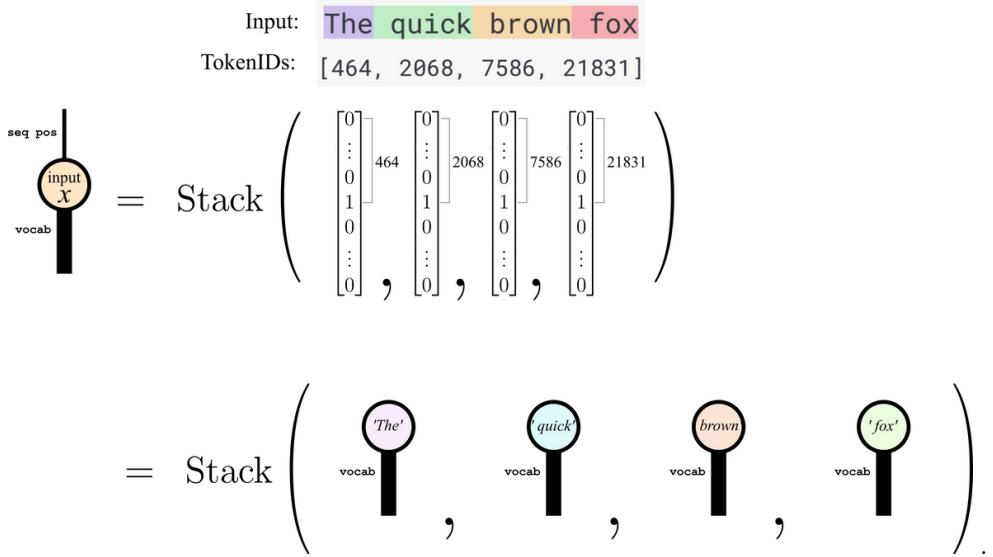


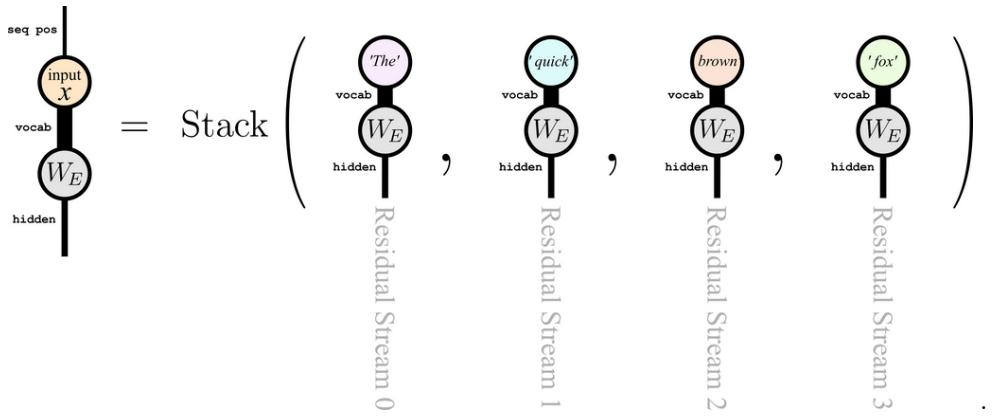
FIG. 4: A tensor network diagram of GPT-2. Computationally, the network is contracted from top to bottom (input at the top, output at the bottom). Here we have shown one of the 12 layers of attention and MLP blocks, but the other layers are just repetitions of the attention and MLP blocks shown above, but with different learned weight parameters  $W_Q, W_K, W_V, W_O, W_\uparrow, W_\downarrow$ .

but here's what it looks like regardless:



The output tensor  $y$  (found by contracting the input with the whole network) has the same shape, but in general every entry will be nonzero, as it represents the log probability that the model predicts for every possible next token at every position. The vector at the final token position represents the log probability distribution for the unknown next word, which can be sampled to turn the predictive model into a generative model.

The embedding layer is responsible for getting the input into the model, by compressing (projecting) the input vector on each token from a  $\text{dim}(\text{vocab}) \approx 50,000$  dimensional space down to a  $\text{dim}(\text{hidden}) \approx 768$  dimensional space known as the residual stream of each token:

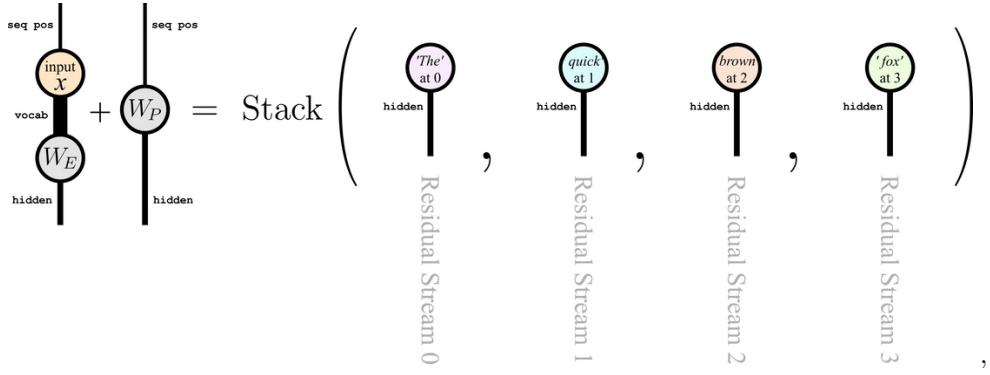


The total dimension of the residual stream is  $\text{dim}(\text{seq pos}) \times \text{dim}(\text{hidden})$ : there is one stream for each input token, and each token's stream is of dimension  $\text{dim}(\text{hidden})$ .

Presumably this process of embedding involves creating a vector representing the token's meaning independent of any context from the tokens around it, by packing similar tokens into similar parts of residual-stream space (though [superposition](#) may also be involved).

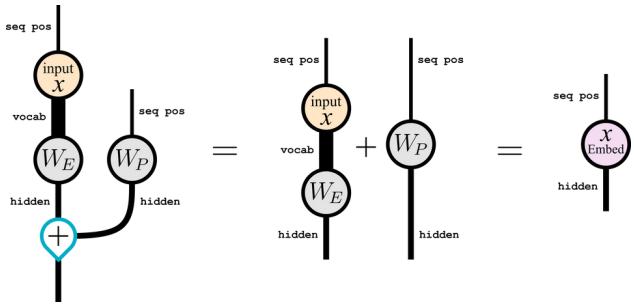
The embedding also has another component: the positional embedding. This is a simple fixed vector for each position, added to each residual stream independent of the token at that position.

It lets each residual stream have some information about where in the sequence it is located:



as long as a subspace in the residual stream is reserved to store and use this positional information.

However different transformer architectures tend to use very different kinds of positional embedding techniques, so for simplicity throughout the rest of the sections we'll contract the input and the embeddings into a single tensor:



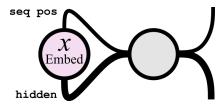
but feel free to substitute in the expanded form whenever you like.

The unembedding matrix works similarly, but without any positional component.

The structure of the transformer from here is a series of “blocks” which copy from, then add back into the residual stream. There are two kinds of blocks: Attention and MLP, which alternate down through the layers.

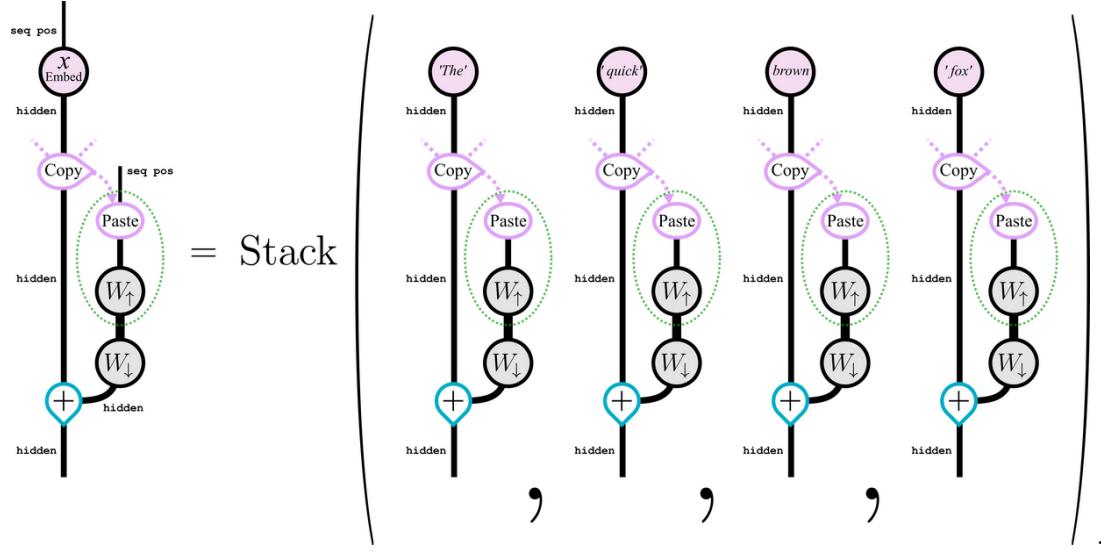
## B. MLP

The MLP layer is most like a dense neural network, though it is residual (copying from and adding into the residual stream, rather than modifying it directly), and acts only on the `hidden` index. Unlike an MLP layer, a general transformation would be able to move information between token positions, by also acting on the `seq pos` index like so:



but MLP layers in transformers only involve contractions onto the `hidden` leg, and so act independently on each token, meaning that they can't move information from one residual stream to

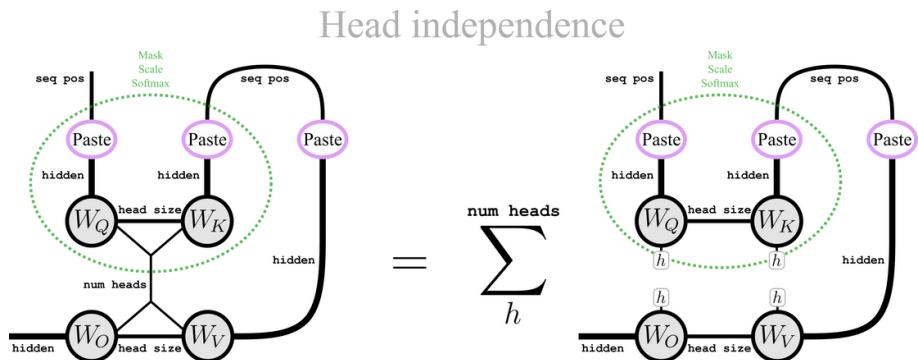
another:



Despite acting independently on each residual stream, MLPs make up the vast majority of the parameters in a transformer because they project up to a higher-dimensional space with the  $W_\uparrow$  matrix<sup>14</sup> before applying a GELU nonlinearity and projecting back down again with  $W_\downarrow$ . The MLP parameters in  $W_\uparrow$  and especially  $W_\downarrow$  seem to be where most of the trained facts and information are stored, as evidenced by [Transformer Feed-Forward Layers Are Key-Value Memories](#) [31], the [ROME \(Rank One Model Editing\)](#) paper [32] and subsequent work on [activation patching](#) for locating and editing facts inside transformers.

### C. Attention

Each attention layer consists of a number of heads, which are responsible for moving relevant data from the residual stream of preceding tokens, transforming it, and copying it into the residual stream of later tokens. Each head acts effectively independently - we can see this in our diagram by replacing the `num heads` leg with a sum, and seeing that each term contributes independently to the residual stream:

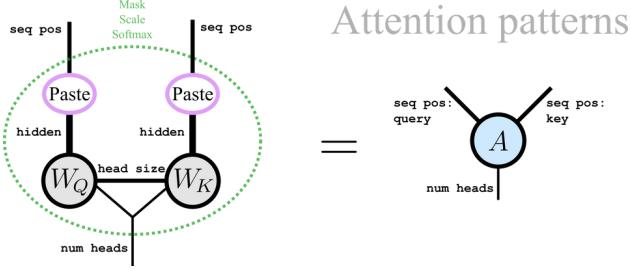


Attention heads host the most easily interpretable parts of a transformer: the attention patterns. These are low-rank matrices (one matrix for each attention head, as indexed by the `num heads`

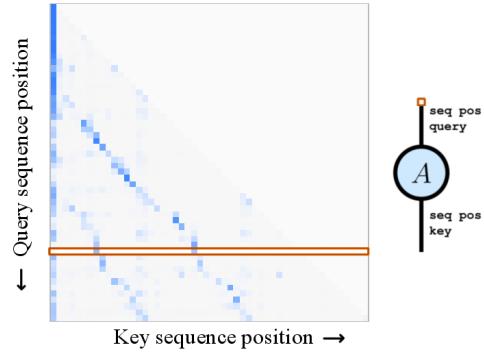
---

<sup>14</sup> a  $4 \times$  higher dimensional space in the case of GPT-2

leg) calculated like so:



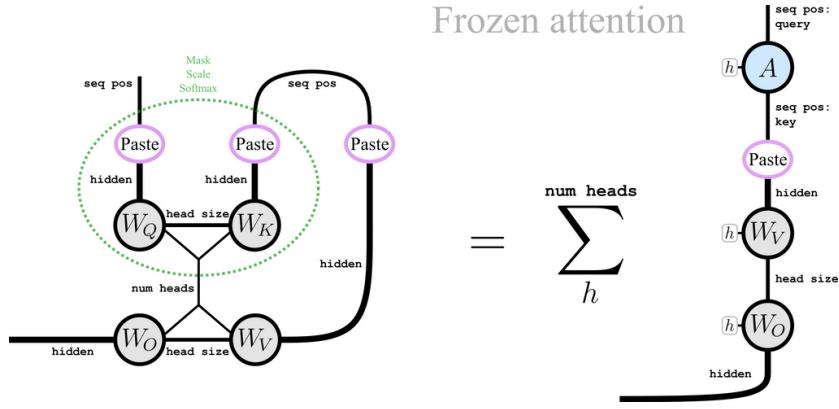
Attention patterns determine how information is moved between tokens. For example, this attention pattern seems to move information from earlier tokens matching a pattern onto tokens immediately preceding the equivalent token in a new language:<sup>15</sup>



<EOT> EN: This is the largest temple that I've ever seen.  
FR: C'est le plus grand temple que j'ai jamais vu.  
DE: Das ist der größte Tempel, den ich je gesehen habe.

You can play with [the interactive version](#) of this in the [In-context Learning and Induction Heads paper](#). [33] We'll see an example of how an attention pattern like this can come about in section IX, but for now we can just take attention patterns for granted.

Rather than computing attention patterns on the fly based on the current context in the residual stream (the two “pasted” tensors in pink), the attention pattern can be “frozen” for easier interpretability, fixing the  $A$  tensor and therefore fixing a specific pattern of information movement. When this is done, the attention block simplifies to



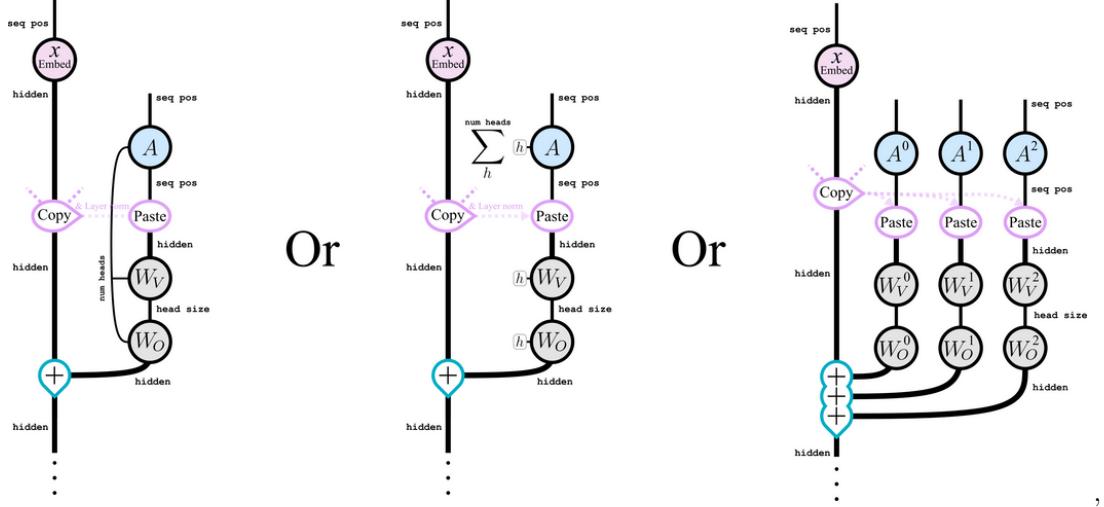
which is completely linear: the only non-einsum operation is now just a single copy and paste from the residual stream.<sup>16</sup> We can also see that the attention pattern  $A$  is the only transformation in

<sup>15</sup> This specific “induction head” kind of attention pattern will only be seen after the first layer, because it must arise as a result of composition with attention head(s) in previous layers.

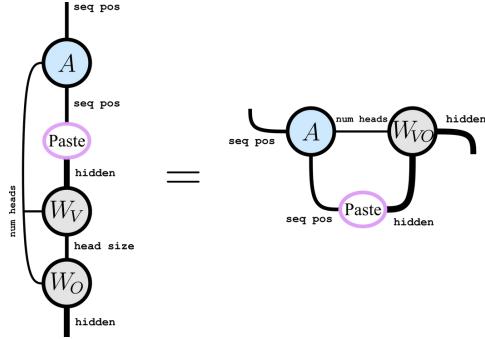
<sup>16</sup> Copying and pasting can in general be nonlinear, for example if products are taken between copies of an object, that's the same as raising it to some power: a nonlinear operation. But here with attention frozen there are no products taken between copies: just a sum when the attention result is added back into the residual stream.

the whole network which ever acts on the `seq pos` index: every other tensor is contracted into the `hidden` index, so every other linear transformation can be described independently for each token. This is why the attention pattern is the only part of the network that can move data between tokens.

With attention frozen, we can represent the sum over heads in a number of ways:



these are all equivalent, just placing different emphasis on the independence of each head. You can even go in the opposite direction and emphasize the matrix-multiply nature of the `num heads` leg, reminiscent of low rank decompositions like SVD but for operators:<sup>17</sup>



though this low-rank-operator way of looking at attention is probably less useful than the sum-of-heads way of looking at it.

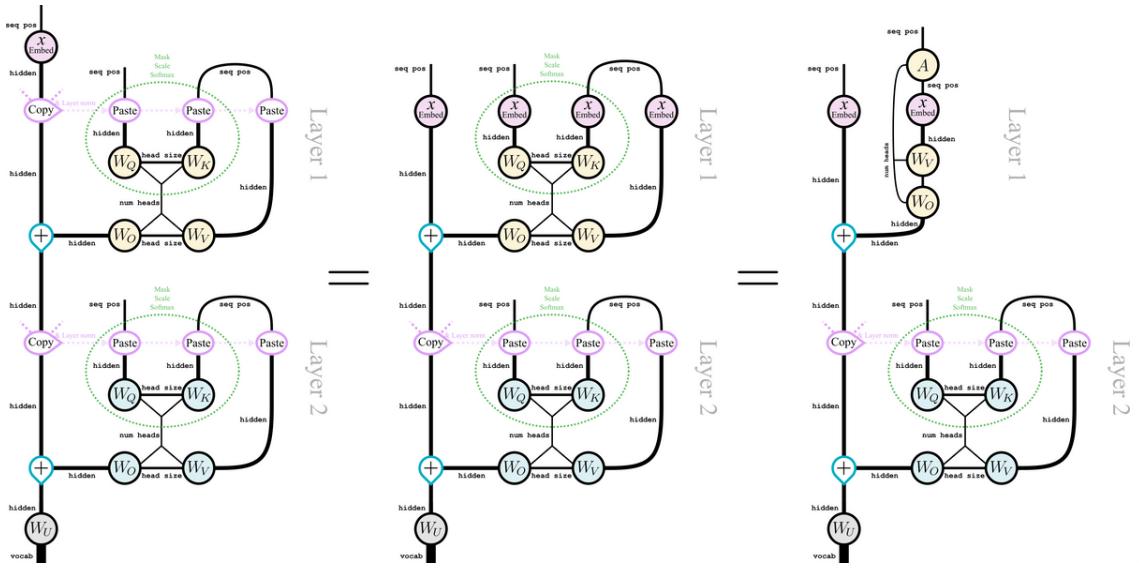
---

<sup>17</sup> This is reminiscent of other decompositions of tensor operators into sums of rank-one tensor products, such as sums of strings of single-site Pauli operators in quantum error correcting codes, and Matrix Product Operators (MPOs) more generally.

## VIII. COMPOSITION AND PATH EXPANSION

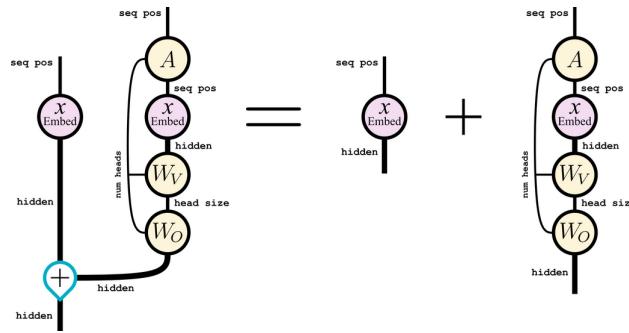
Layers of Attention and MLP blocks don't just act in isolation: they all copy from and write to the same residual stream, so later layers can use information computed in earlier ones. Still following [A Mathematical Framework for Transformer Circuits](#), [4] I'll ignore MLP blocks and focus only on the ways that attention heads in an earlier layer can "compose" with those in a later layer. I'll also ignore small but annoying nonlinearities like [layer normalization](#).

So here's a two layer attention-only transformer:



where in the middle we've gone ahead and copied the input into the first attention layer, and on the right we've contracted the attention patterns in the first layer.

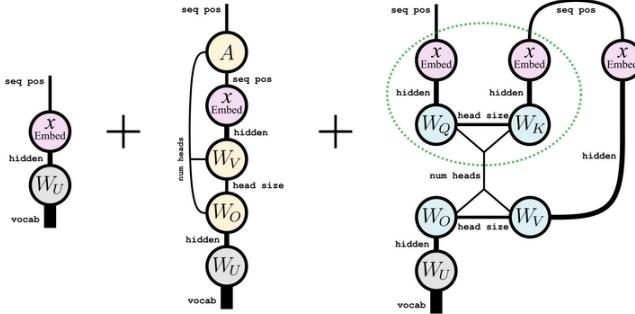
When the second layer copies from the residual stream, it will copy the a sum of terms from earlier layers. Rather than treating the result of this sum as a single complicated object, we can keep the sum expanded as two separate terms: the original input  $x$  plus a "perturbation" caused by the first layer:



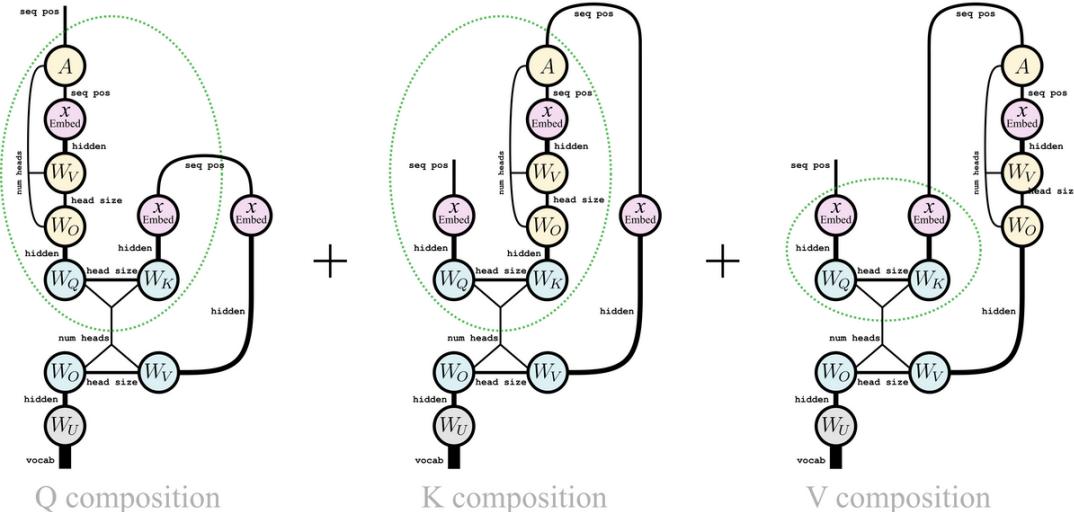
Now we can expand the output of the whole network as a sum of terms like this, in something

reminiscent of a [perturbation theory](#):

### Non-composition terms:



### + Single compositions:

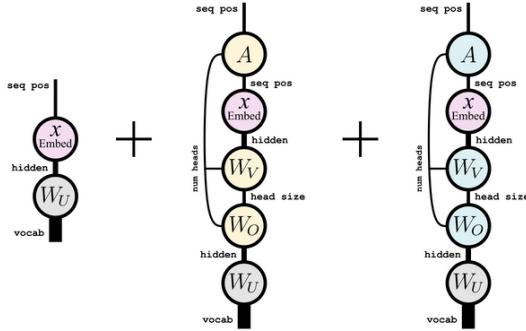


### + Higher order terms (double and triple compositions)

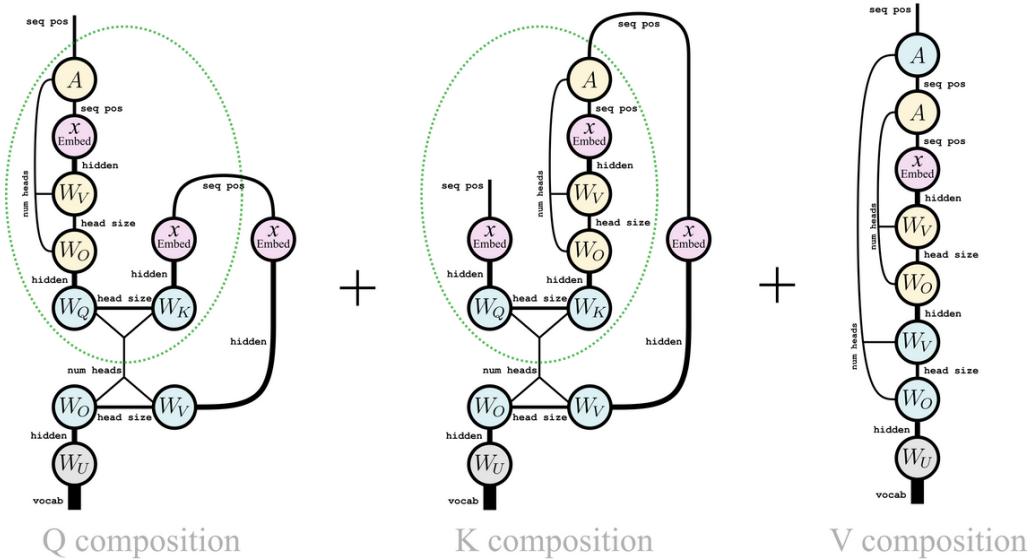
Note: this expansion is not technically true as written due to the nonlinearities (green ellipses) not acting independently on each term in the sum: this is what it means to be nonlinear. However these nonlinearities are pretty linear - consisting only of a mask (which is linear), a re-scale, and a softmax to ensure that the elements in each row of the attention pattern add to 1. The layer normalization nonlinearities (not shown) [can also largely be treated as approximately linear](#) with sufficient care. [4] So we see that there are three simplest kinds of nontrivial attention composition: Q-composition, K-composition, and V-composition. We can simplify the V-composition and non-composition terms slightly by noticing when the attention patterns in the second layer can also be

frozen:

### Non-composition terms:



### + Single compositions:



### + Higher order terms (double and triple compositions)

so we can see that V-composition has a simpler iteration structure than Q or K composition - the attention pattern is just formed by matrix-multiplying two attention patterns, and likewise for the  $W_O$  and  $W_V$  transformations.

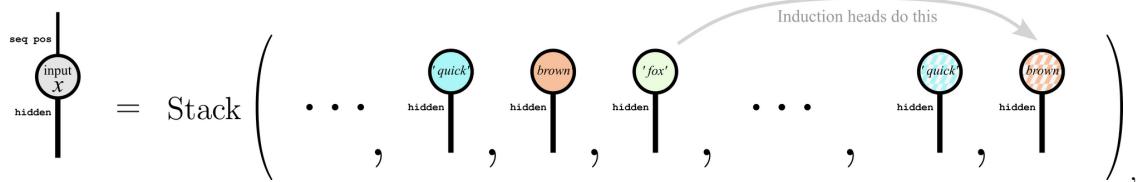
There are ten terms which contribute to the final answer for two layers: three non-compositions (shown), three single compositions (shown), and four higher-order compositions (not shown). But we can (and usually should) also expand out the `num heads` indices as sums, and have a separate term for each head and combination of heads. Likewise for the `seq pos` indices if we want to consider the contributions to or from specific token positions. [MLP layers could also be incorporated into this expansion sum](#), [4] though their stronger nonlinearities would require some kind of linearization, and they only have one kind of nontrivial composition anyway. Regardless, the number of terms grows exponentially with the number of layers, so this kind of trick will only be useful if we have some reason to think that most of the work is being done by relatively few terms (preferably low order ones).

Some intuition for thinking that relatively few terms are important comes from noticing that each head can only write to a relatively small subspace of the residual stream, because the `head size` dimension is small compared to the `hidden` dimension, and so each head is putting a relatively small dimensional vector into a relatively high dimensional space with  $W_O$  and selecting small parts of a high dimensional residual stream with  $W_Q$  and  $W_K$ , leaving sufficient room in the

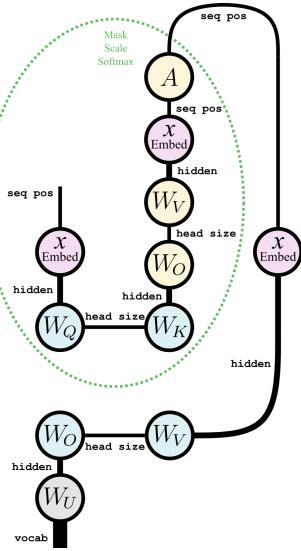
residual stream for most heads in different layers to act mostly independently if they want.<sup>18</sup> Of course it's an empirical matter if low order terms *actually* explain most of the relevant behaviors, so this should be [checked empirically](#). There may also be much more effective ways of decomposing the computations of a transformer into a series of terms or circuits like this, such that more of the relevant behavior is explained by fewer more interpretable terms. One recent method attempting to find them is the [Automated Circuit DisCovery \(ACDC\)](#) algorithm. [34]

## IX. EXAMPLE: TOY INDUCTION HEAD

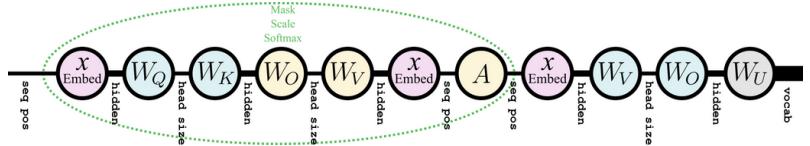
We'll finish off by constructing one toy example of an induction head: a circuit detecting what should come next in a repeated string of tokens. For example, consider predicting the next word in some text like "The quick brown fox [...] the quick brown". It seems like "fox" should come next to fit the pattern. Information from the earlier "fox" token should be copied into the final "brown" token's residual stream, so that the model can predict "fox" for the next word there:



This is known as induction, which is a type of in-context learning. There are many ways to make these induction heads, and real induction heads are likely to be messy, but things resembling them [have been found](#) in models of virtually all sizes. [33] We'll construct a handcrafted toy example of an induction circuit, by forming a "virtual induction head" from the K-composition of two heads in different layers:



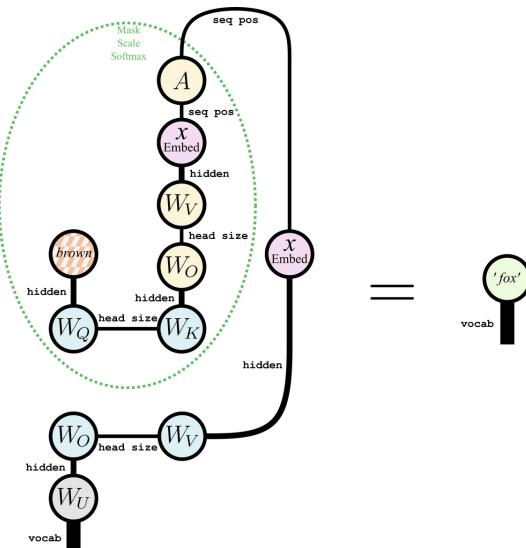
Everything in this diagram is now a matrix, so we could draw it to emphasize that it's just a sequence of matrix multiplications:



<sup>18</sup> Though the same isn't true for MLP layers. Additionally, if `num heads × head size ≈ hidden dim` (as is usually true), and the contribution of each head to its subspace is not small, then some decent number of heads per layer **must** interact with heads in the previous layer.

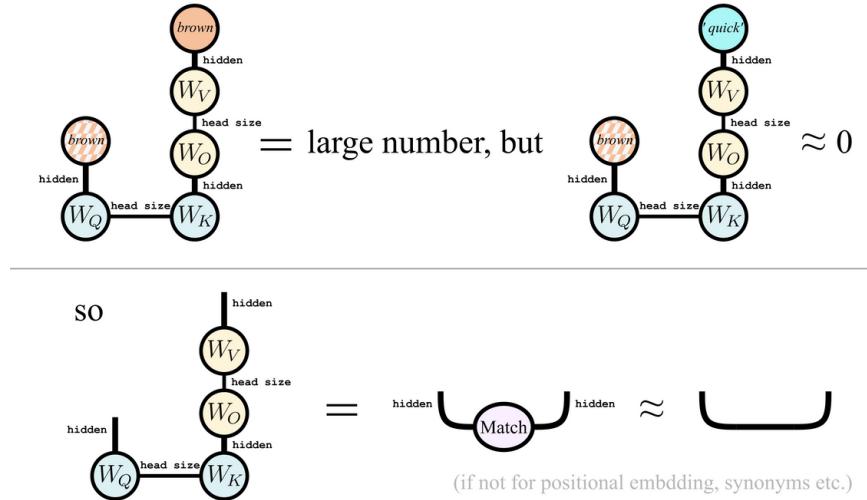
but we'll stick with the previous format so you can more easily see how the circuit fits into the rest of the network.

In order for this to act like an induction head, the last “brown” input token (brown) should output a “fox” token:<sup>19</sup>



Induction involves exploiting repeated patterns, so induction heads had better be able to pattern match. Here's the subcomponent of our induction circuit which will do that:

Pattern matching: we want



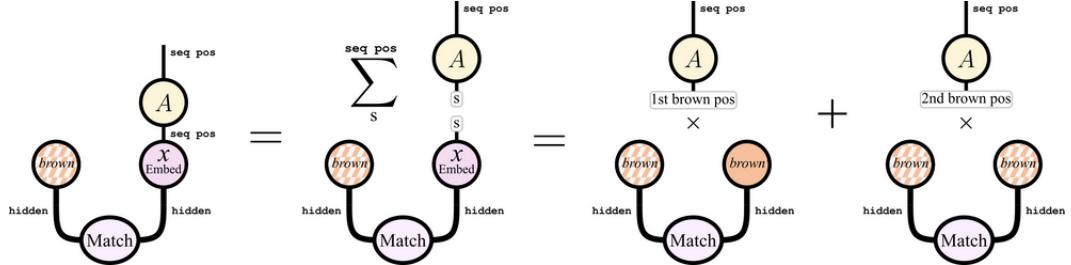
So  $W_O$  and  $W_V$  in layer 1 (shown in yellow) and  $W_Q$  and  $W_K$  in layer 2 (shown in blue) can effectively just behave like identity matrices in the relevant subspace, or anything else producing something like a “semantic delta tensor” when you compose them together. We denote this tensor as “Match” because it should be near zero when contracted with any two vectors unless the vector on the left semantically “matches” with the vector on the right.

Now, we don't really care about the matching token per se, we just want to know which token came after it. We can see that the “key” side of the attention pattern  $A$  from layer 1 is going to be

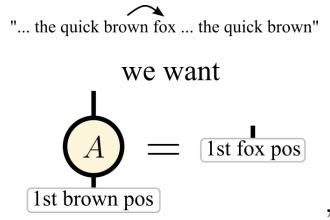
---

<sup>19</sup> The original “brown” vector will probably also have to be subtracted out of the residual stream somewhere too. This is also ignoring positional embeddings.

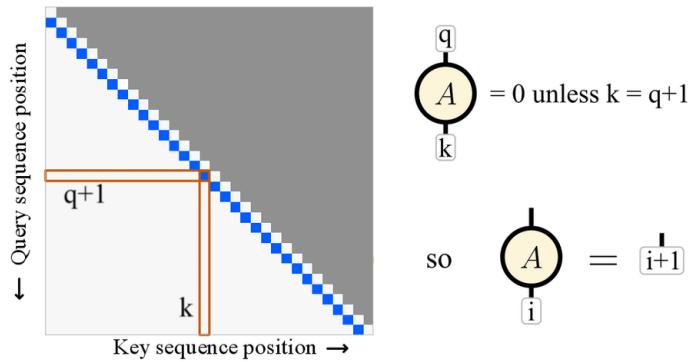
indexed at whatever token positions we get a match on (and multiplied by a number depending on how strong the match is):



We know that the token we want is in the position directly after `1st brown pos`, so  $A$  should map an index of `pos` on its key side to `pos + 1` on its query side so that we can index that token. Then we'll have

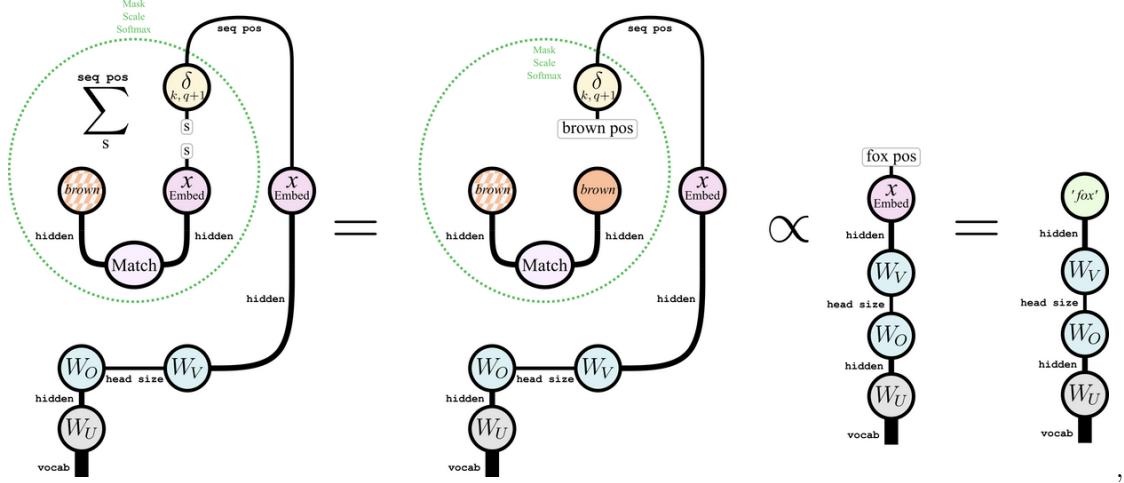


as we desire. We see that an attention pattern  $A$  doing this can just be a fixed off-diagonal delta-tensor  $A_{k,q} = \delta_{k,q+1}$ :



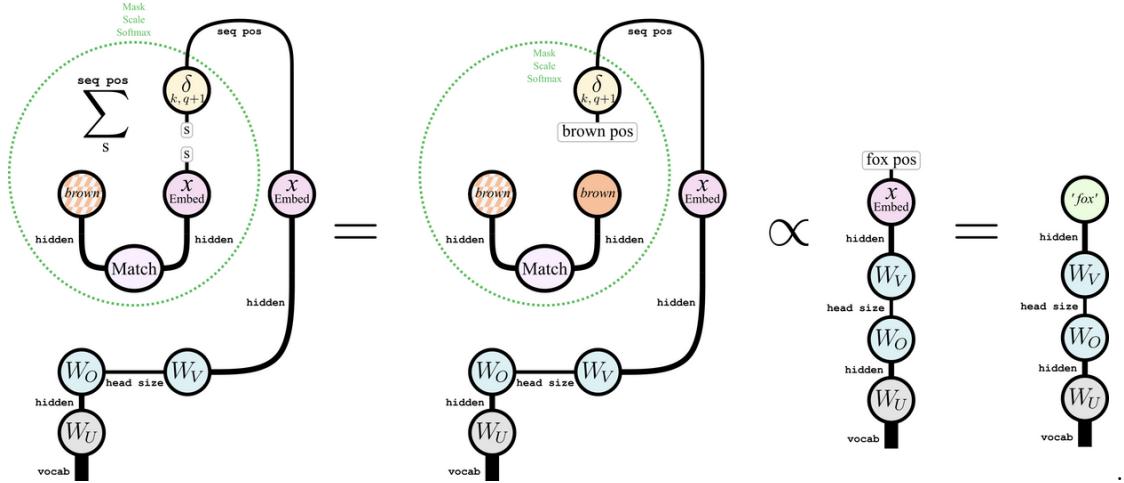
This is also equivalent to mapping `pos` on its query side to `pos - 1` on its key side: always just attending to the previous token, so heads with this attention pattern are known as “previous token heads”. This attention pattern also removes the unwanted `2nd brown pos` term, because this term indexes the final column of the attention matrix where all entries are zero.

Putting this attention pattern  $A_{k,q} = \delta_{k,q+1}$  into our circuit therefore simplifies it to:



so we can get the desired “fox” token out, so long as it gets properly handled by  $W_V$  and  $W_O$  (which are in charge of putting the “fox” information into the residual stream), and by the unembedding  $W_U$  (which is in charge of getting the “fox” information out of the residual stream and turning it into the correct “fox” token).

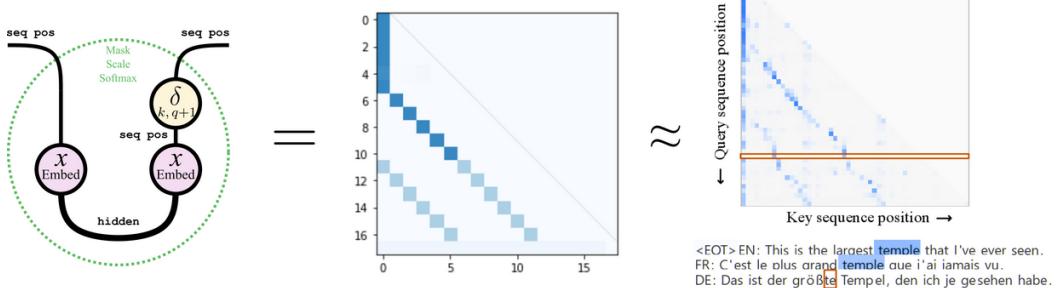
Putting everything together, we see that it’s possible for virtual induction heads to have a very simple approximate form, composed almost entirely of delta-like tensors:



We can sanity check this toy induction head by computing its attention pattern numerically on a repeating sequence of random vectors. We see that it looks like a real induction-head pattern,

attending to the tokens which followed the current token previously in the sequence:

### Induction head sanity check:



Where  $x$  is a three times repeated sequence of random vectors:

$$\text{seq pos} \begin{array}{c} \text{Embed} \\ \text{hidden} \end{array} = \text{Stack} \left( \begin{array}{c} \text{random vector } 0 \\ \text{hidden} \end{array}, \begin{array}{c} \text{random vector } 1 \\ \text{hidden} \end{array}, \begin{array}{c} \text{random vector } 2 \\ \text{hidden} \end{array}, \dots, \begin{array}{c} \text{random vector } 0 \\ \text{hidden} \end{array}, \begin{array}{c} \text{random vector } 1 \\ \text{hidden} \end{array}, \begin{array}{c} \text{random vector } 2 \\ \text{hidden} \end{array}, \dots, \begin{array}{c} \text{random vector } 0 \\ \text{hidden} \end{array}, \begin{array}{c} \text{random vector } 1 \\ \text{hidden} \end{array}, \begin{array}{c} \text{random vector } 2 \\ \text{hidden} \end{array}, \dots \right),$$

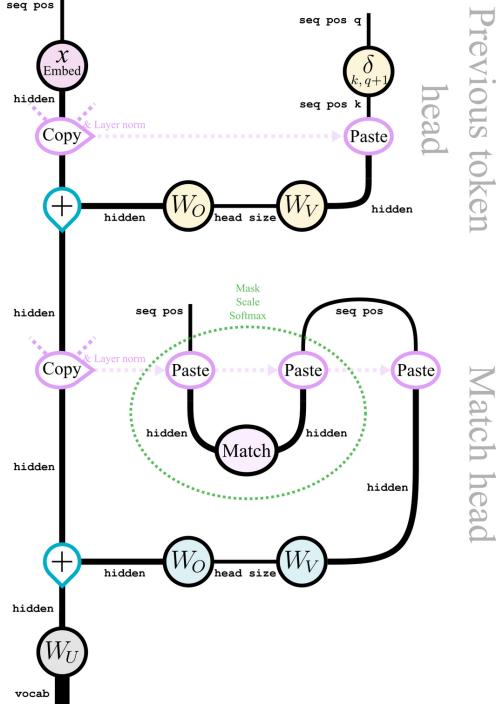
here's the code for that:

```
import torch as t
import matplotlib.pyplot as plt
from einops import einsum, repeat
hidden_dim = 768
pattern_len = 6
# Generate a three times repeated sequence of 6 random vectors
x = t.rand((pattern_len, hidden_dim))
x = repeat(x, 'seq hidden -> (repeat seq) hidden', repeat=3)
seq_len = x.shape[0]
# Calculate the toy attention pattern
prev_token_head = t.diag(t.tensor([1.0]*(seq_len-1)), diagonal=-1)
match = t.eye(hidden_dim)
attn_pattern = einsum(x, match, x, prev_token_head, 'seq0 hidden0, hidden0
    hidden1, seq1 hidden1, seq1 seq2 -> seq2 seq0')
# Apply the mask and softmax to the attention pattern
attn_pattern = (t.tril(attn_pattern)-t.triu(t.ones_like(attn_pattern)*1e5))
attn_pattern = attn_pattern.softmax(dim=-1)
print('attn_pattern = ')
plt.imshow(abs(attn_pattern), cmap='Blues'); plt.show()
```

However this virtual induction head formed by K-composition is just one term in the path expansion, so we'd need to make sure that this is actually the dominant term by suppressing the others. This is where the  $W_{Q/K/O/V/U}$  matrices are important, because they selectively determine what gets taken from and added into the residual stream, allowing them to suppress unwanted terms in the path expansion (if they so choose) so that only this virtual induction head is important.

Finally, we can put these  $W_{O/V/U}$  matrices back in and see the full induction network rather

than just the “virtual attention head” term in the path expansion:



## X. CONCLUSION

Overall graphical tensor notation is a useful tool for understanding and communicating interpretability results, especially for anything involving operations between more than a few tensors. It surfaces dualities and interesting equivalences more easily than other notation, and remains intuitive without necessarily losing any mathematical rigor. It may just be my preference, but I frequently run into papers where I wish this notation was used.

## ACKNOWLEDGEMENTS

I thank my PhD advisor Ian McCulloch for introducing me to tensor networks and the associated graphical notation, and for much support in working with these things over the years. Thanks also to the attendees and mentors at the 2nd [Machine Learning for Alignment Bootcamp \(MLAB\)](#), and to authors of excellent existing explanations of graphical tensor notation, such as in [the math3ma blog](#), [Simon Verret's blog](#), [tensoरnetwork.org](#), [tensors.net](#), and [Hand-waving and Interpretive Dance: An Introductory Course on Tensor Networks](#). This work was supported by an Australian Government Research Training Program (RTP) Scholarship, and by [AI Safety Support](#) through the [ML Alignment & Theory Scholars Program \(MATS\)](#).

*Editable SVG files for all diagrams in this document are available at <https://drive.google.com/drive/folders/16uuvkG1NtjhAbchJ2R1Zp96IoZYKtk5>.*

---

[1] Miles Stoudenmire, Paul Springer, Andrzej Chuchmala, Shentago Jiang, Stefano Crotti, Benjamin Decker, Yu-Hang Tang, Tsuyoshi Okubo, and Pavel Stishenko. the Tensor Network, Jul 2023. URL [https://tensoরnetwork.org](https://tensoरnetwork.org). [Online; accessed 6. Jul. 2023].

- [2] Roger Penrose. Applications of Negative Dimensional Tensors. *Combinatorial Mathematics and its Applications*, pages 221–244, 1971.
- [3] Johnnie Gray. quimb: A python package for quantum information and many-body calculations. *Journal of Open Source Software*, 3(29):819, September 2018. ISSN 2475-9066. doi:10.21105/joss.00819.
- [4] Nelson Elhage, Neel Nanda, Catherine Olsson, Tom Henighan, Nicholas Joseph, Ben Mann, Amanda Askell, Yuntao Bai, Anna Chen, Tom Conerly, Nova DasSarma, Dawn Drain, Deep Ganguli, Zac Hatfield-Dodds, Danny Hernandez, Andy Jones, Jackson Kernion, Liane Lovitt, Kamal Ndousse, Dario Amodei, Tom Brown, Jack Clark, Jared Kaplan, Sam McCandlish, and Chris Olah. A mathematical framework for transformer circuits. *Transformer Circuits Thread*, 2021. <https://transformer-circuits.pub/2021/framework/index.html>.
- [5] Tai-Danae Bradley. Matrices as Tensor Network Diagrams, Jun 2021. URL <https://www.math3ma.com/blog/matrices-as-tensor-network-diagrams>. [Online; accessed 18. Jun. 2021].
- [6] Simon Verret. Tensor network diagrams of typical neural networks. *Simon Verret's*, Feb 2019. URL <https://simonverret.github.io/2019/02/16/tensor-network-diagrams-of-typical-neural-network.html>.
- [7] Glen Evenbly. Tensors.net, July 2023. URL <https://www.tensors.net>. [Online; accessed 6. Jul. 2023].
- [8] Jacob C. Bridgeman and Christopher T. Chubb. Hand-waving and interpretive dance: an introductory course on tensor networks. *J. Phys. A: Math. Theor.*, 50(22):223001, May 2017. ISSN 1751-8113. doi: 10.1088/1751-8121/aa6dc3.
- [9] Yao Lei Xu, Kriton Konstantinidis, and Danilo P. Mandic. Graph Tensor Networks: An Intuitive Framework for Designing Large-Scale Neural Learning Systems on Multiple Domains. *arXiv*, March 2023. doi:10.48550/arXiv.2303.13565.
- [10] Lam Chi-Chung, P. Sadayappan, and Rephael Wenger. On Optimizing a Class of Multi-Dimensional Loops with Reduction for Parallel Execution. *Parallel Process. Lett.*, 07(02):157–168, Jun 1997. ISSN 0129-6264. doi:10.1142/S0129626497000176.
- [11] Carsten Damm, Markus Holzer, and Pierre McKenzie. The complexity of tensor calculus. *Comput. Complexity*, 11(1):54–89, June 2002. ISSN 1420-8954. doi:10.1007/s00037-000-0170-4.
- [12] Callum McDougall. Six (and a half) intuitions for SVD, July 2023. URL <https://www.lesswrong.com/posts/iupCxk3ddiJBAJkts/six-and-a-half-intuitions-for-svd>. [Online; accessed 1. Feb. 2024].
- [13] Erhard Schmidt. Zur Theorie der linearen und nichtlinearen Integralgleichungen. *Math. Ann.*, 63(4): 433–476, December 1907. ISSN 1432-1807. doi:10.1007/BF01449770.
- [14] Carl Eckart and Gale Young. The approximation of one matrix by another of lower rank. *Psychometrika*, 1(3):211–218, September 1936. ISSN 1860-0980. doi:10.1007/BF02288367.
- [15] L. Mirsky. SYMMETRIC GAUGE FUNCTIONS AND UNITARILY INVARIANT NORMS. *Q. J. Math.*, 11(1):50–59, January 1960. ISSN 0033-5606. doi:10.1093/qmath/11.1.50.
- [16] Frank L. Hitchcock. The Expression of a Tensor or a Polyadic as a Sum of Products. *J. Math. Phys.*, 6(1-4):164–189, April 1927. ISSN 0097-1421. doi:10.1002/sapm192761164.
- [17] J. Douglas Carroll and Jih-Jie Chang. Analysis of individual differences in multidimensional scaling via an n-way generalization of “Eckart-Young” decomposition. *Psychometrika*, 35(3):283–319, September 1970. ISSN 1860-0980. doi:10.1007/BF02310791.
- [18] Richard A. Harshman. Foundations of the PARAFAC procedure: Models and conditions for an “explanatory” multi-model factor analysis. *UCLA Working Papers in Phonetics*, 16:1–84, 1970.
- [19] Ledyard R. Tucker. Some mathematical notes on three-mode factor analysis. *Psychometrika*, 31(3): 279–311, September 1966. ISSN 1860-0980. doi:10.1007/BF02289464.
- [20] Johan Håstad. Tensor rank is NP-complete. *J. Algorithms*, 11(4):644–654, December 1990. ISSN 0196-6774. doi:10.1016/0196-6774(90)90014-6.
- [21] M. Fannes, B. Nachtergaele, and R. F. Werner. Finitely correlated states on quantum spin chains. *Commun. Math. Phys.*, 144(3):443–490, March 1992. ISSN 1432-0916. doi:10.1007/BF02099178.
- [22] A. Klümper, A. Schadschneider, and J. Zittartz. Groundstate properties of a generalized VBS-model. *Z. Phys. B: Condens. Matter*, 87(3):281–287, October 1992. ISSN 1431-584X. doi: 10.1007/BF01309281.
- [23] Stellan Östlund and Stefan Rommer. Thermodynamic Limit of Density Matrix Renormalization. *Phys. Rev. Lett.*, 75(19):3537–3540, November 1995. ISSN 1079-7114. doi:10.1103/PhysRevLett.75.3537.
- [24] Guifré Vidal. Efficient Classical Simulation of Slightly Entangled Quantum Computations. *Phys. Rev. Lett.*, 91(14):147902, October 2003. ISSN 1079-7114. doi:10.1103/PhysRevLett.91.147902.
- [25] Ian P. McCulloch. From density-matrix renormalization group to matrix product states. *J. Stat. Mech.: Theory Exp.*, 2007(10):P10014, October 2007. ISSN 1742-5468. doi:10.1088/1742-5468/2007/10/P10014.
- [26] I. V. Oseledets. Tensor-Train Decomposition. *SIAM J. Sci. Comput.*, September 2011. URL <https://epubs.siam.org/doi/abs/10.1137/090752286>.
- [27] M. B. Hastings. An area law for one-dimensional quantum systems. *J. Stat. Mech.: Theory Exp.*, 2007(08):P08024–P08024, Aug 2007. ISSN 1742-5468. doi:10.1088/1742-5468/2007/08/p08024.
- [28] Lluís Masanes. Area law for the entropy of low-energy states. *Phys. Rev. A*, 80(5):052104, Nov 2009. ISSN 2469-9934. doi:10.1103/PhysRevA.80.052104.

- [29] Itai Arad, Alexei Kitaev, Zeph Landau, and Umesh Vazirani. An area law and sub-exponential algorithm for 1D systems. *arXiv*, Jan 2013. URL <https://arxiv.org/abs/1301.1162v1>.
- [30] Anurag Anshu, Itai Arad, and David Gosset. An area law for 2D frustration-free spin systems. *arXiv*, Mar 2021. URL <https://arxiv.org/abs/2103.02492v2>.
- [31] Mor Geva, Roei Schuster, Jonathan Berant, and Omer Levy. Transformer feed-forward layers are key-value memories. In Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih, editors, *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 5484–5495, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. doi:10.18653/v1/2021.emnlp-main.446. URL <https://aclanthology.org/2021.emnlp-main.446>.
- [32] Kevin Meng, David Bau, Alex Andonian, and Yonatan Belinkov. Locating and editing factual associations in GPT. *Advances in Neural Information Processing Systems*, 36, 2022.
- [33] Catherine Olsson, Nelson Elhage, Neel Nanda, Nicholas Joseph, Nova DasSarma, Tom Henighan, Ben Mann, Amanda Askell, Yuntao Bai, Anna Chen, Tom Conerly, Dawn Drain, Deep Ganguli, Zac Hatfield-Dodds, Danny Hernandez, Scott Johnston, Andy Jones, Jackson Kernion, Liane Lovitt, Kamal Ndousse, Dario Amodei, Tom Brown, Jack Clark, Jared Kaplan, Sam McCandlish, and Chris Olah. In-context learning and induction heads. *Transformer Circuits Thread*, 2022. <https://transformer-circuits.pub/2022/in-context-learning-and-induction-heads/index.html>.
- [34] Arthur Conmy, Augustine N. Mavor-Parker, Aengus Lynch, Stefan Heimersheim, and Adrià Garriga-Alonso. Towards Automated Circuit Discovery for Mechanistic Interpretability. *arXiv*, April 2023. doi:10.48550/arXiv.2304.14997.