Chapter 7
Introduction to Transformer Interpretability

Contents

1 Introduction

Transformer mechanistic interpretability aims at reverse engineering transformer models inner functioning and providing human understandable explanations.

This document requires readers to be familiar with the transformer architecture. If this is not the case, taking the time needed to read the dedicated section 2 or the *Illustrated Transformer* blog post [31] is recommended.

Papers are presented in both chronological and increasing order of difficulty. Our exploration will begin with the *Logit Lens* [21], an interpretability tool that allows us to examine how transformers refine their predictions of the next token across layers. Following that, *Locating and Editing Knowledge in Transformers* [16] will be presented, a paper that introduces a method for localizing and modifying precise pieces of knowledge in a transformer.

Transformer mechanistic interpretability relies heavily on a series of papers known as the *Transformer Circuits Thread*. It was inspired by the original *Thread: Circuits* [33] that focuses on reverse engineering vision models. Some of the papers from the *Transformer Circuits Thread* will be introduced:

- *A Mathematical Framework for Transformer Circuits* [2] investigates toy transformer models, comprising up to two layers. In particular, the authors identify an interesting type of circuit, the induction heads, that could explain the ability of transformer for in-context learning (the ability to use information from far back in the context to predict the next token of the sequence). This work is fundamental for understanding subsequent papers in the series.
- *In-Context Learning and Induction Heads* [24], the second paper in the series by Olsson et al., delves deeper into induction head circuits, and offers compelling arguments suggesting they might be the primary mechanism driving in-context learning.
- *Softmax Linear Units* [11] is the third paper in the series. It introduces a new activation function, SoLU, that enhances the interpretability of transformers MLP layers, that were previously shown to be highly polysemantic and difficult to understand. This paper will also be the occasion to introduce the concepts of polysemanticity and superposition, which are fundamental in mechanistic interpretability.
- *Towards Monosemanticity: Decomposing Language Models With Dictionary Learning* finally tackles MLP layers and suggests to train an autoencoder to decompose features learned by MLP layers.

In Section 9, the paper *Interpretability in the Wild* will be discussed. It provides a detailed analysis of how GPT-2 small performs the task of identifying indirect objects. Finally, concept erasure, which refers to the deliberate removal of certain concepts or knowledge from a model's learned parameters, is introduced through a paper named *LEACE: Perfect Linear Concept Erasure in Closed Form.*

This document is a work in progress that does not yet mention important works. Additional sections will be added in the future.

2 High-level Structure of Transformers

The transformer architecture was introduced in the historical paper *Attention is All You Need* [36]. If you don't have a clear picture of how a transformer works, it is recommended to spend time reading and deeply engaging with this section. The Mathematical Framework presents the transformer architecture from a distinct viewpoint compared to Jay Alammar's approach [1].

The high-level architecture of a transformer can be divided into three fundamental components (see Figure 1):

1. A **token embedding**
2. A series of **residual blocks**, each of them composed of
   (a) An **attention layer** (also called multi-head attention layer), which comprises multiple attention heads, denoted as $h_i \in H$. These attention heads function independently and in parallel. Their outputs get aggregated at the end of the attention layer.
   (b) An **MLP layer**, denoted m.
3. A **token unembedding**

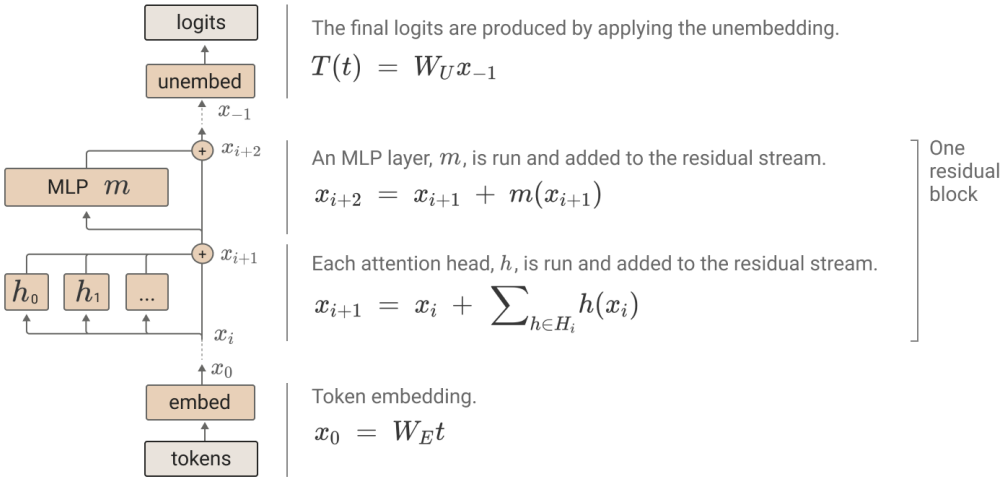

Figure 1: High-level architecture of a transformer. This figure should be read from bottom to top. The transformer receives a vector of tokens as input and outputs a vector score over all possible tokens (the logits). In between, there is a stack of residual blocks, each consisting of an attention head and an MLP layer. It's important to note that this representation of a transformer is somewhat unconventional: in general, focus is put on the attention heads and MLPs, here the residual connection between the embedding and unembedding appears in the middle. With the Logit Lens (Section), it has been seen previously that there is a lot to understand about transformers by looking at the information flowing through this residual connection [2].

[1] Note that the models investigated in *Mathematical Framework* are decoder-only transformers, whereas the original *Attention is All You Need* paper and Jay Alammar's blog post introduce both decoder-and-encoder transformers.

Notations and variables

$n_{context}$: context size, i.e. maximum number of tokens in the context window (GPT-4 biggest version has a context size of 32,768 tokens).

n_{vocab}: vocabulary size (typically $\sim 50,000$).

d_{model}: dimension of the model, i.e. size of the residual stream (in large models it can go into the tens of thousands)

d_{head}: dimension of the attention heads (often 64 or 128).

d_{mlp}: number of hidden neurons in each MLP (typically 4 times bigger than d_{model})

2.1 The Residual Stream

The transformer's residual stream is a central object. All layers communicate through it but it does not do any processing of its own. It can thus be thought of as a **communication channel**. Attention blocks and MLP layers "read in" information from the residual stream as input, internally transform this information, and subsequently "write" back into the residual stream (it's not exactly writing, it's more like adding). Each layer makes an incremental modification to the residual stream.

Figure 2: Transformer residual stream [2].

Let's explain more precisely now Figure 1. The shapes of parameters are indicated within brackets.

1. The token embedding, denoted as x_0 $[n_{context}, d_{model}]$, serves as the model's input. It is computed by multiplying the token embedding matrix, W_E $[d_{model}, n_{vocab}]$, with the one-hot encoded tokens, t $[n_{context}, n_{vocab}]$. You can think of the token embedding as a tensor with one embedding per row.

2. In the first residual block, the attention block "reads in" the token embedding x_0.

   (a) Each attention head (in the first layer H_0) denoted as h operates on this embedding, and their individual outputs are all summed up and "written" back into the residual stream, yielding a new embedding, $x_1 = x_0 + \sum_{h \in H_0} h(x_0)$.

   (b) Then, the MLP layer within the first residual block "reads in" the embedding x_1 from the residual stream and outputs the transformed embedding $m(x_1)$ back into the residual stream. This process results in a new embedding, $x_2 = x_1 + m(x_1)$.

3. This same process repeats for each subsequent residual block: attention heads and MLP layers successively "read in" from the residual stream, x_n $[n_{context}, d_{model}]$, and "write" into it, x_{n+2} $[n_{context}, d_{model}]$.

4. Finally, the final embedding is transformed back into logits via multiplication with the unembedding matrix, W_U $[n_{vocab}, d_{model}]$. The overall transformer function is denoted as T.

The residual stream size is $[n_{context}, d_{model}]$: it is a high-dimensional vector space. Comparatively, attention heads get much smaller inputs $[n_{context}, d_{head}]$, with $d_{head} << d_{model}$. This means that attention heads operate on small subspaces of the residual stream. Some layers read in and write on disjoint subspaces and thus never "interact". Once added, information persists in a subspace unless another layer actively deletes it: this is why the residual stream can be seen as a form of memory or bandwidth.

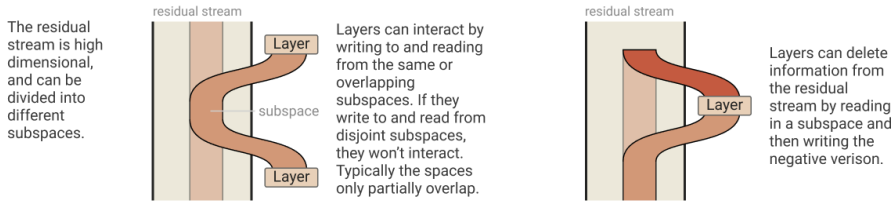

Figure 3: The residual stream can be thought of as a bandwidth. It stores information and moves it between layers. Note that the residual stream size is d_{model}, while the attention heads size is $d_{head} < d_{model}$. This is why layers read and write in subspaces of the residual stream [2].

2.2 The Attention Head Layer

Attention heads are integrated within residual blocks (see Figure 1). Within a single residual block, there are multiple attention heads, usually 8 or 12, which operate *independently* and *in parallel*.

Now let's look into these attention heads.

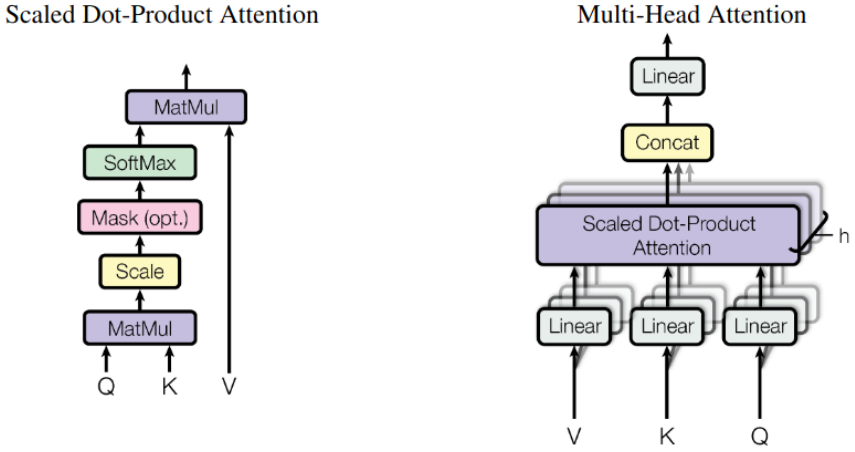

Figure 4: Attention heads mechanism as described in the original transformer paper [36].

Here is how attention heads operate:

1. In a given layer, all **attention heads read in from the residual stream**, denoted here x $[n_{context}, d_{model}]$.
2. **Each attention head independently computes three vectors**: a *query vector* Q, a *key vector* K, and a *value vector* V, all with dimensions $[n_{context}, d_{head}]$. Q is computed as the dot product between a learned query matrix W_Q and the embedding vector x : $Q = W_Q x$ (same for $K = W_K x$ and $V = W_V x$). It's important to note that each attention head learns its own set of parameter matrices; it would be more rigorous to denote them as W_Q^h, W_K^h, and W_V^h. All have dimensions $[d_{head}, d_{model}]$.
3. Next, these three vectors are used to generate a *result vector* r. It is computed as follows:

$$r = softmax(\frac{Q^T K}{\sqrt{d_{head}}})V$$

   The $Q^T K$ matrix is called the *attention pattern*. It can be useful to observe it to get an understanding of what the head is doing.
4. The outputs from all attention heads are concatenated into a single array $[r^{h_1}, r^{h_2}, ...]$, which is then multiplied by an output matrix W_O^H. This output matrix can be virtually divided into individual components as $W_O^H = [W_O^{h_1}, W_O^{h_1}, ...]$. The output of an attention layer is thus:

$$[W_O^{h_1}, W_O^{h_1}, ...][r^{h_1}, r^{h_2}, ...]^T = \sum_i W_O^{h_i} r^{h_i}$$

3 The Logit Lens

The *Logit Lens* [21] is an interpretability tool designed to explore how the residual stream (also known as hidden activations) of transformers evolves across layers. Since understanding the transformer is crucial to understanding the *Logit Lens* the following section provides a quick recap of the core elements of the transformer architecture and how the *Logit Lens* applies to it, before the subsequent section delves into the *Logit Lens* observations.

3.1 Transformer and Logit Lens Overview

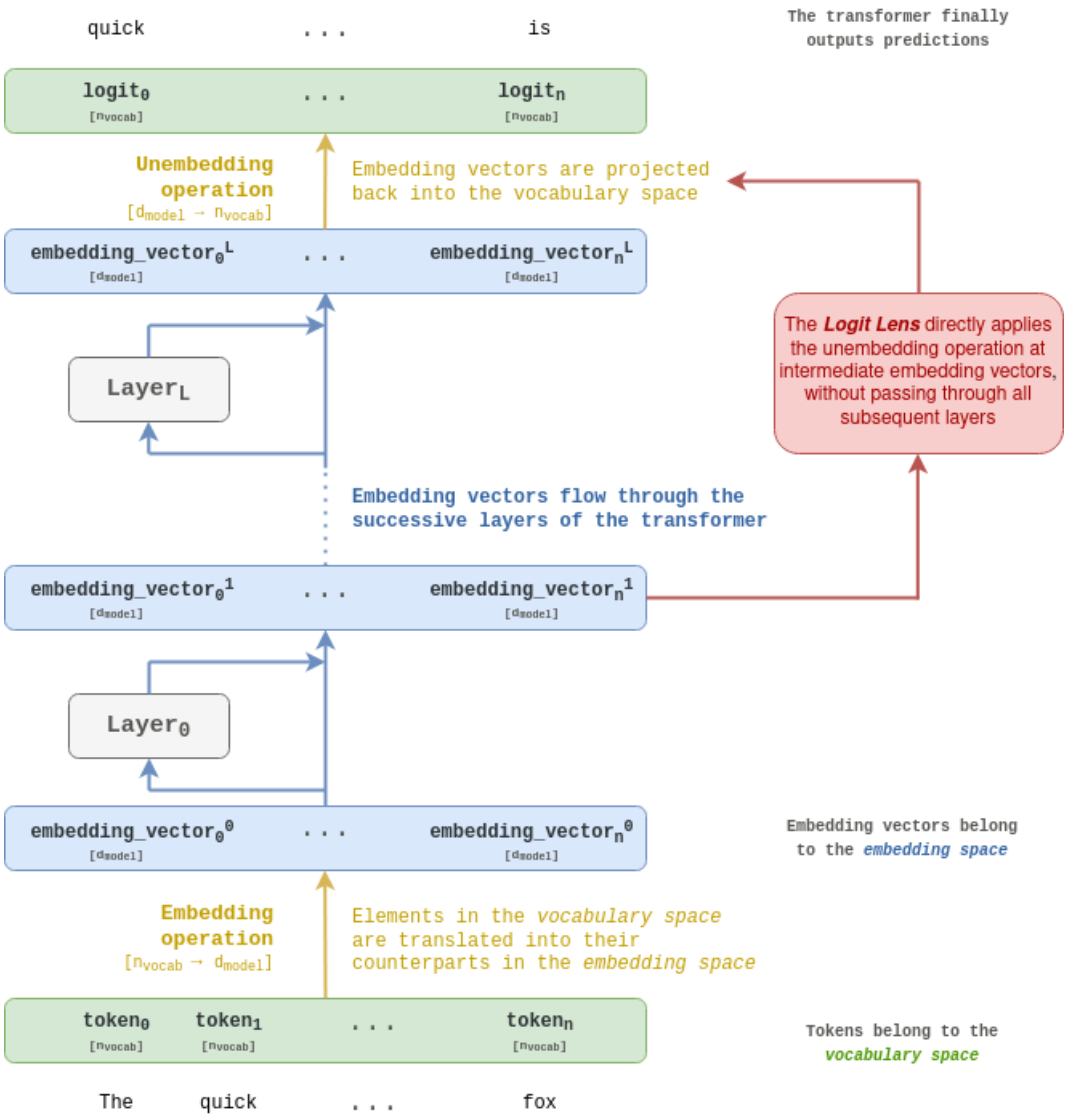

Figure 5: Overview how how the *Logit Lens* works. Read from bottom to top. A detailed explanation is provided in the following paragraphs.

A transformer takes as an **input** a **sequence of tokens**, or units of text. Each token is an element from a vocabulary (typically of size $n_{vocab} \sim 50,000$), so each text token can be associated with an integer value ranging from 0 to n_{vocab}. This association can also be thought of as representing the token as a basis vector within a space of dimension n_{vocab}, denoted the **vocabulary space**. Each basis vector in the vocabulary space is associated with one token.

A transformer is trained to predict the next token of the input sequence, so its **output** is a **probability distribution over the vocabulary**.

Transformers convert each tokens into vectors living in a space with a much lower dimensionality than the vocabulary size n_{vocab}, which is known as the **embedding space**. The embedding space dimension is denoted d_{model}, so $d_{model} < n_{vocab}$. Projected tokens are called **embedding vectors**, or simply *embeddings*. In some contexts, they may also be referred to as *hidden activations*. The **embedding matrix** used to convert tokens from the vocabulary space to the embedding space can be seen as a lookup table that associates elements in the vocabulary space with their corresponding counterparts in the embedding space.

Once tokens have been converted into the embedding space, they **flow through the successive layers of the transformer**. Similar to how input images undergo transformations across the layers of a Convolutional Neural Network (CNN), embedding vectors are transformed as they pass through the network. The **Logit Lens** precisely focuses on those intermediate representations that transformers build.

Finally, to output a probability distribution over the vocabulary, the final embedding vector has to be converted back into vocabulary space. Conversion from the embedding space to the vocabulary space can be done through multiplication by a matrix of dimension $[d_{model}, n_{vocab}]$. This operation is called the **unembedding** [2]. The result of the unembedding is a *logits vector* of size n_{vocab}. The logits can then be converted into probabilities over the vocabulary using the softmax function.

The essential intuition behind the *Logit Lens* is that the unembedding operation, typically applied after the last layer, can also be applied after each intermediate layer. Embedding vectors are modified by each layer of the network, but their dimensions remain the same (d_{model}). Thus, after any layer, embedding vectors can be converted back to the vocabulary space, and one can get an idea of how the network refines its prediction across layers.

3.2 Observations

Fig 6 shows how using the *Logit Lens* looks in practice.

[2] Note however that the embedding and unembedding matrices are not necessarily the transpose of one another !

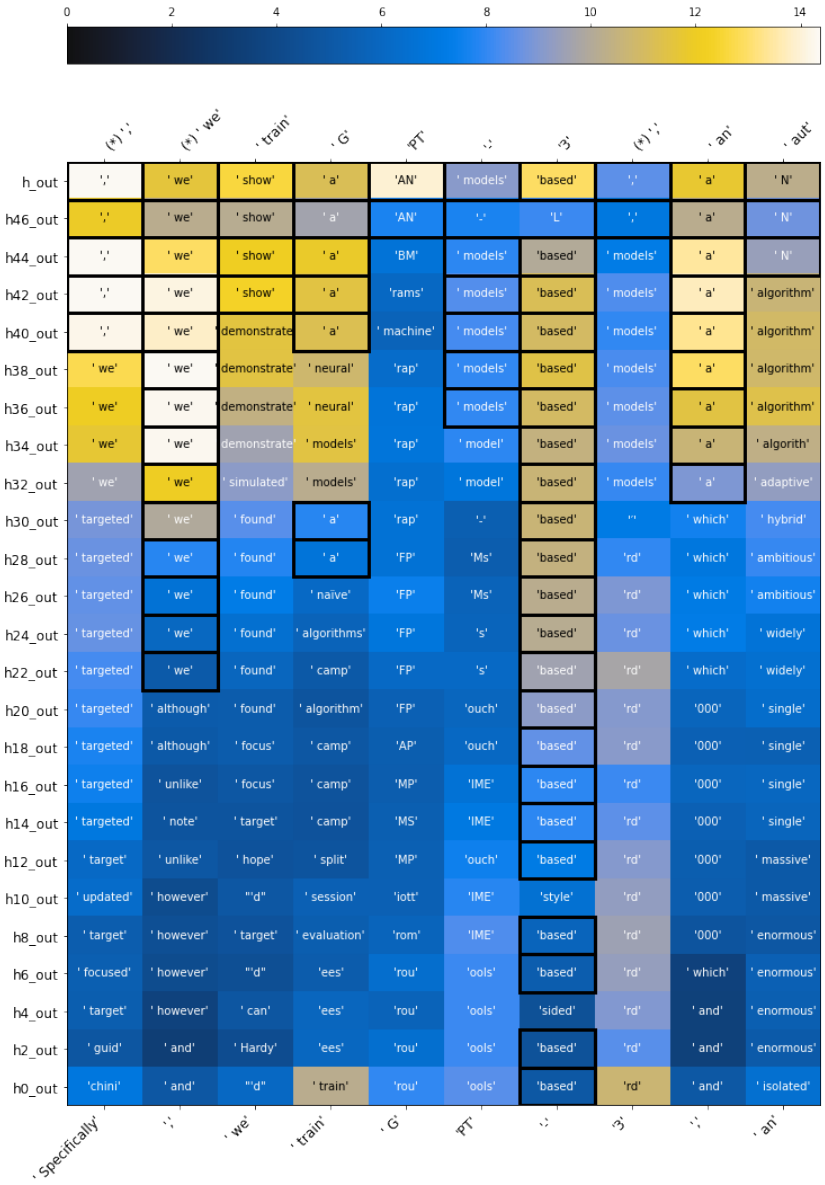

Figure 6: An example of the *Logit Lens* used on GPT-2. Input tokens are written at the bottom, and correct outputs at the top. Layers stack from bottom to top. The first token at the top, ",", corresponds to the token the model should predict when given as input only the first token "Specifically". The second token at the top, "we", corresponds to the token the model should predict when given as input only the two first tokens "Specifically, ". This is the reason why there is a shift of one position between tokens at the bottom and tokens at the top. (*) indicates that the model correctly predicted the next token. As an example, to find the word "session" in cell ["h10_out", "train"], the embedding of the vector "train" after layer 10 (size d_{model}) was multiplied by the unembedding matrix (size $[d_{model}, n_{vocab}]$). This multiplication generated a logit vector of size n_{vocab}. The highest entry in the logit vector corresponds to the token "session". It's the token that the model judges to be the most likely to follow the input "Specifically, we train". Each cell thus contains the model's top guesses at different latent steps. The color scale indicates the associated logit value. The higher the logit, the more confident in its prediction the model is [21].

The key observations and intuitions that can be drawn from the *Logit Lens* are the following:

- Transformers seem to incrementally refine their guess of the next token.
- The trajectory of latent prediction smoothly converges to the final output distribution. Logits values associated with latent predictions increase pretty smoothly across layers.

- Instead of "keeping in memory" the input tokens over the first layers, transformers seem to immediately convert them into a very different representation, which is then progressively refined. The input token (under the matrix) and the prediction after layer 0 ("h0_out") are often semantically very different.

Fig 7 is an other example of how the transformer predicts a sequence with many repetitions: "I love plasma, I love plasma, I love plasma, etc.". "Plasma" is quite a rare token, which is not very likely to occur after the sequence "I love". Here, the author wanted to see at what point the repeating pattern is "noticed" by the network. Predicting the first occurrence of "plasma" from "I love" should be complicated, but as the network is provided with more repetitions of "I love plasma" in its input, it should become easier and the network should get more confident when predicting "plasma". It seems that the repetition is noticed in the upper half of the network, but not in the lower half, even after several rounds of repetition.

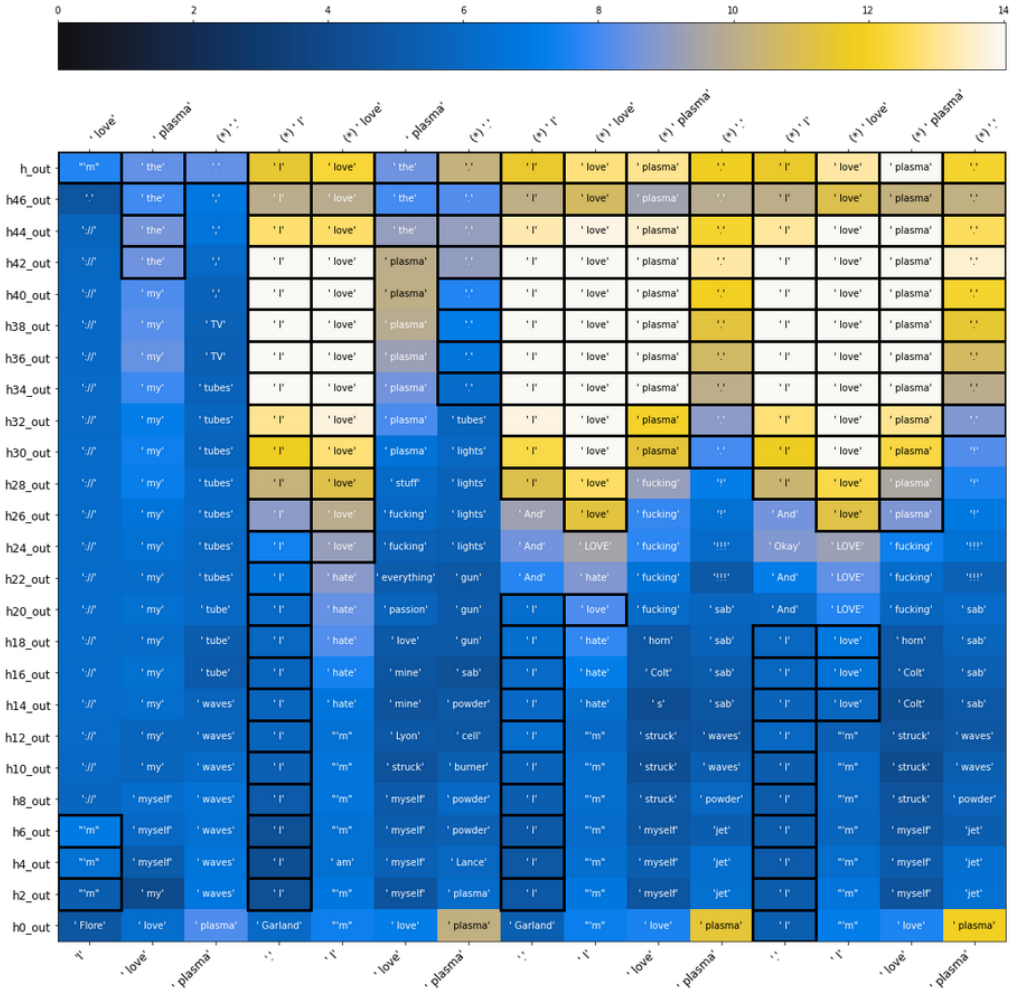

Figure 7: GPT-2's top tokens and their logits on a repeating sequence [21].

Note that a refined version of the *Logit Lens*, the *Tuned Lens* [4], was developed subsequently to cope with some of its limitations. In particular, the *Logit Lens* yields plausible results on GPT-2 but fails to elicit predictions from several subsequent models. It also systematically puts more probability mass on certain vocabulary items than the final layer does.

The difference between the *Tuned Lens* and the original *Logit Lens* is that instead of applying the unembedding matrix, an affine transformation is trained for each layer to translate representations from the basis of each layer to the basis expected at the final layer.

4 Locating and Editing Knowledge in Transformers

In *Locating and Editing Factual Associations in GPT* [16] Meng et al. aim to localize and modify knowledge within transformers. They manage to teach networks counterfactual statements such as "The Eiffel Tower is located in the city of **Rome**" by slightly adjusting the weights of a single MLP matrix.

More precisely, they focus on locating and editing *factual associations*. Factual associations refer to statements like "The Space Needle is in downtown Seattle", characterized by a subject ("The Space Needle"), an object ("Seattle"), and the relation between them ("is in downtown").

Knowledge editing holds great potential, as it can be used to fix networks when they are incorrect, biased, or reveal private information.

To locate factual knowledge, Meng et al. introduce a novel method known as **causal tracing**, which identifies the MLP activations that have the strongest impact on prediction. After knowledge is located within MLP activations, they edit the models' weights using a second method called **ROME (Rank-One Model Editing)**.

4.1 Causal Tracing Method to Locate Knowledge

The causal tracing method aims to identify which part of the network is involved when the network recalls a fact. It involves three key steps:

1. **Clean run**. **Let's imagine one want to localize where the factual association "The Space Needle is in downtown Seattle" is stored in a network**. The first thing to do is simply to prompt the network: "The Space Needle is in downtown". It will then likely output the correct token: "Seattle". All activations from this forward pass are saved, because they will be needed during the final step (see Fig 8(a) Clean run).

2. **Corrupted subject run**. For the second step, a second forward pass is performed with the same input: "The Space Needle is in downtown", except that this time, noise is added to the tokens corresponding to the subject: "The Space Needle"[3]. Intuitively, from the perspective of the network, it is as if it had received something like: "<noise> <noise> <noise> <noise> is in downtown". It will likely output something wrong because important information was lost (see Fig 8(b) Corrupted subject run).

3. **Corrupted-with-restoration run**. Now, a final forward pass is performed, with the same input as in the second step (the noisy one). Remember that at step 1, transformer's activations were saved, they will be needed here. Take a look at the purple arrow on Fig 8. Residual stream activations (also called hidden states) corresponding to the token "Need", at layer 2, are copy-pasted from the clean run to this last run, in the middle of the forward pass. This operation "injects" clean information somewhere in the model. If information about the factual association between "The Space Needle" and "Seattle" is located in this part of the residual stream, then pasting the clean run activations is like information retrieval and helps the network finding the right answer. This final step is repeated for each token and each network subcomponent (residual stream, MLPs, and attention heads), leading to Fig 9.

[3] More precisely, noise is introduced into activations immediately after the token embedding.

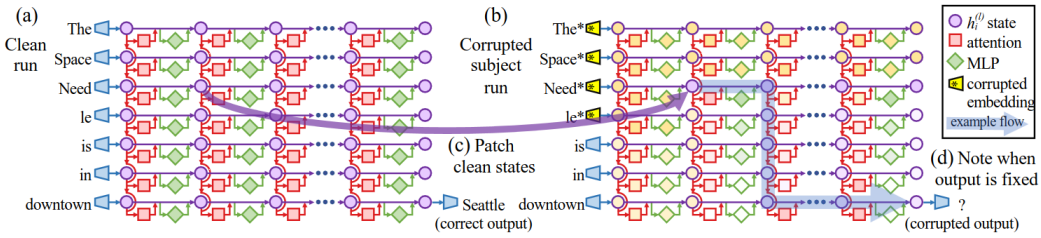

Figure 8: The causal tracing method isolates the causal effect of individual components within the network and enables to identify where in a network a certain piece of information is located. Layers are stacked from left to right. Embeddings are in blue, hidden states (residual stream) in purple, attention heads in red and MLP layers in green [16].

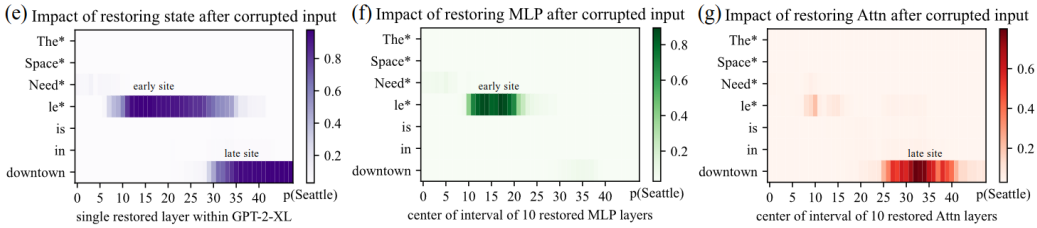

Figure 9: Effect of restoring individual model components on the probability of outputting the correct token. (e) illustrates the effect of restoring hidden states (residual stream) on the prediction, (f) the effect of restoring MLPs, and (g) the effect of restoring attention heads' activations. Meng et al. identify two sites with significant causality: the early site, in the middle layers residual stream and MLPs of the last subject token, and the late site, in the final layers and the last token [16].

The presence of the late site is not surprising. It's pretty obvious that restoring the residual stream of the final token in the final layers will make prediction easier. However, the presence of the early site is a discovery of the paper. It indicates that a relatively small set of MLPs in transformers' mid-layers contain the information on the factual association, and that this information is "retrieved" only on the last token of the subject (and not uniformly on all subject's tokens).

Meng et al. end up proposing the **localized factual association hypothesis**, which posits that middle layers' MLPs serve as a storage and retrieval mechanism for factual associations. Factual associations are made of a subject, an object, and the relationship between them.

According to their hypothesis, mid-layer MLPs, when they receive as input subject's tokens, generate meaningful representations of it across their layers, to finally output memorized properties or facts associated with that subject. Middle layer MLPs collectively accumulate information related to the subject, which is then transferred to the last token in the sequence[4].

4.2 ROME Method to Edit Knowledge

The ROME (Rank-One Model Editing) method is designed to update the weights of a transformer's MLP layer and modify a factual association stored in it. In general it only requires the modification of one single MLP layer.

[4]It may be interesting to note attention heads' role is to move information between tokens [2]. Attention heads might be responsible for the transfer of information to the last token.

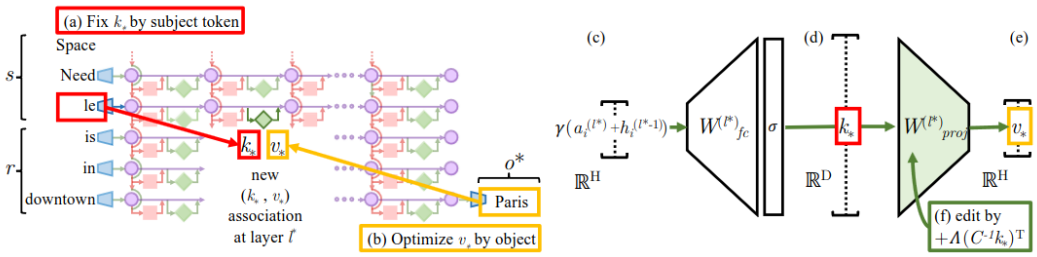

Figure 10: The Rome paper makes the assumption that the information about the subject's properties is stored in the second matrix (green on the diagram here). This is obviously an important assumption (and one that is also largely false). But once this assumption has been made, all we have to do is adapt the green matrix in such a way that it maps the subject k* with the desired properties v*, by applying a small deviation of rank 1 to the W matrix. The vector k* can be found by taking the average embedding of sentences containing the subject "Eiffel Tower". And to find v* , simply begin by taking a random differentiable embedding v, and perform a gradient ascent on v to transform v into v* to maximise the logit of the desired token [16].

MLP layers can be modeled as *two-layer key-value memories*, where the first layer forms a key, and the second layer retrieves an associated value. Let's denote $K = [k_1, k_2, ...]$ the matrix of key vectors and the corresponding value matrix as $V = [v_1, v_2, ...]$. The MLP can be seen as an application W that maps keys to values: $WK \approx V$. You can think of a key as representing the subject (ex: "The Space Needle"), and of a value as representing the object and its relation to the subject (ex: "is in downtown Seattle")[5].

To insert a new key-value association $(k*, v*)$ the steps are to (i) determine the vector $k*$ that represents the subject, (ii) determine the vector $v*$ so as to maximize the recalling of the new object, (iii) and finally update the MLP weights W.

$k*$ is found by taking a set of sentences ending with the subject and averaging the activations collected at the layer that will be updated. For example: ""

$v*$ is found through gradient ascent on the embedding space to maximize the probability of the new fact being recalled.

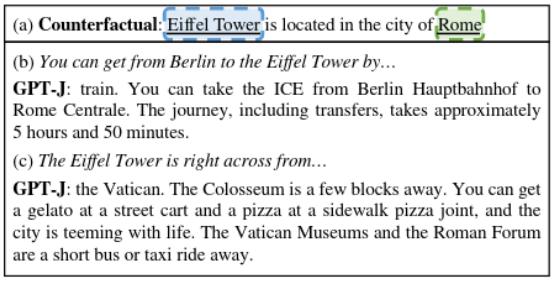

Figure 11: GPT-J was taught the counterfactual "The Eiffel Tower is located in the city of Rome". This new fact seems to generalize across different contexts [16].

The new inserted association must demonstrate both generalization and specificity. **Generalization** measures to what extent knowledge remains robust across different contexts. For instance, if a network learn that the Eiffel Tower is in Rome, it should also know that visiting it requires to travel to Rome. **Specificity** ensures that changes in knowledge do not inadvertently impact unrelated facts. For example, after acquiring the information that the Eiffel Tower is in Rome, you should not mistakenly conclude that all tourist attractions are also located in Rome.

Meng et al. observed that edits made in the early site demonstrate both strong generalization and specificity, while changes in the late site fail to exhibit the same level of generalization.

[5]Note that these are not the same keys and values as in the transformer attention heads.

4.3 Conclusion and Limitations

Locating and Editing Factual Associations in GPT suggests that:

1. Factual associations appear to be stored in transformers within mid-layer MLPs, which progressively output relevant information into the residual stream. This information aggregates in the last layers so as to generate relevant tokens related to the mentioned subject.
2. Editing a single MLP seems sufficient for the network to generalize the new knowledge across various prompts.

However, the ROME method has some limitations worth mentioning [13]:

- Edits are unidirectional. For example, “The iconic landmark in Seattle is the Space Needle” is stored separately from “The Space Needle is the iconic landmark in Seattle”, so modifying both requires two edits [32].
- ROME edits token associations, not underlying concepts. For instance, "cheese" and "fromage" are different tokens and editing one doesn't generalize to the other.
- Edits often conduct to over-generalization. The edited model is more likely to mention the associated object when the subject appears in the context than it was before edition (loud facts).

5 A Mathematical Framework for Transformer Circuits

The *Thread: Circuits* [23] focused on how to reverse-engineer vision models [6]. *A Mathematical Framework for Transformer Circuits* [2], published in 2021, aims at understanding in a similar manner the internal computations of transformers.

Given the complexity of state-of-the-art transformer models, Elhage et al. began their exploration by focusing on simplified models, called "toy transformers". They break down these toy transformers into understandable pieces and identify simple algorithmic patterns they seem to perform. This work is considered a foundational step, and many further works built upon the initial insights presented in *Mathematical Framework*. Their toy transformers are models with zero to two layers, that only have attention blocks (in comparison GPT-3 has 96 layers and additional MLP blocks). MLP blocks were left aside because they are significantly more complex to investigate[7].

The authors recorded a series of videos explaining their findings and intuitions, it is recommended to have a look[8]. There's also an other video, by one author, who provides very valuable insights into the paper [9]. For those interested in gaining a deeper understanding, a set of exercises associated with the paper is also available[10].

5.1 Reverse-engineering Results

Mathematical Framework begins by trying to understand the easiest model possible, a zero-layer transformer, and progressively works its way up with one-layer and two-layers transformers. The main results from each model will be presented one after the other.

Zero-Layer Transformers

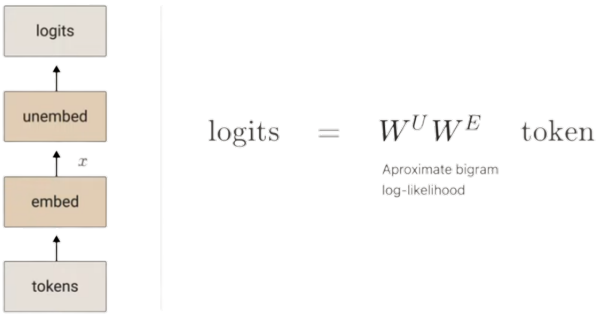

Figure 12: Zero-layer transformer [2].

A **zero-layer transformer** embeds a token and directly unembeds it to produce logits predicting the next tokens : $T = W_U W_E$. As there are no attention heads, information can't flow between tokens. Therefore, the prediction of the next token relies solely on the current token. Zero-layer transformers can be thought of as performing bigram statistics (next token prediction is based on the frequency of two consecutive items occurring in the sequence).

In transformers containing attention heads, it is possible that some tokens may not be affected by attention heads (they only flow through the $W_U W_E$ path). The $W_U W_E$ term can be thought of as modeling bigram statistics, which means that it predicts the likelihood of a word given its preceding word.

[6] See our previous introduction to vision interpretability for more on mechanistic interpretability of vision models.

[7] Layer Normalization and biases were also both left aside. MLP blocks were later investigated in *Towards Monosemanticity: Decomposing Language Models With Dictionary Learning* [34]. See Section 8.

[8] *Transformer Circuits [rough early thoughts]*: https://www.youtube.com/playlist?list=PLoyGOS2WIonajhAVqKUgEMNmeq3nEeM51

[9] A Walkthrough of A Mathematical Framework for Transformer Circuits: https://www.youtube.com/watch?v=KV5gbOmHbjU

[10] Transformer Circuit Exercises: https://transformer-circuits.pub/2021/exercises/index.html

Zero-layer transformers model bigram statistics, and the bigram table can be accessed directly from the weights.

One-Layer Transformers

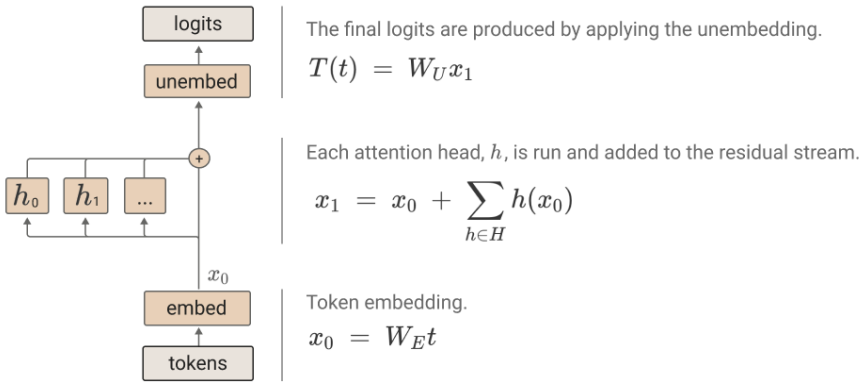

Figure 13: One-layer transformer [2].

A **one-layer transformer** embeds a token, runs a single attention layer and finally unembeds the residual stream.

Operations in this one-layer transformer can be separated into two distinct circuits 14:

- **The Query-Key Circuit (QK circuit, in purple)**, defined by the matrix $W_E^T W_{QK}^h W_E$ $[d_{model}, d_{model}]$, indicates **where the attention head attends to in the context**. It provides the attention score for every pair of (query, key) token. Each entry describes how much a given query token "wants" to attend to and copy information from a given key token.
- **The Output-Value Circuit (OV circuit)**, defined by the matrix $W_U W_{OV}^h W_E$ $[d_{model}, d_{model}]$, describes **what information gets moved from the source token to the destination token in the residual stream**.

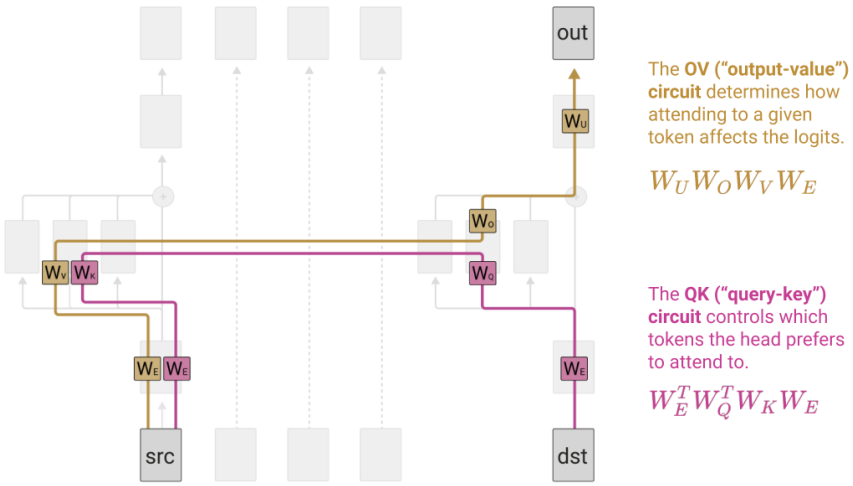

Figure 14: Query-Key and Output-Value Circuits. "src", "dst" and "out" refer to source, destination and out tokens. To understand the behavior of a particular head, a destination token and a source token can be fixed. The output token is the token predicted after the destination token. The value of dst$W_E^T W_{QK}^h W_E$src indicates to what extent the destination token attends to the source token in head h. The value of out$W_U W_{OV}^h W_E$src indicates how much the source token, when the destination token attends to it, affects the probability that the out token will be predicted. The QK circuit determines *where information is moved from and to*, and the OV circuit tells how a given source token impacts the logits when it is attended to [2].

Looking at large values in QK and OV matrices provides insights into the functions performed by attention heads:

- **Primarily positional attention heads**: some attention heads preferentially attend to certain relative positions in the context (for instance, the present or previous token).
- **Copying attention heads**: many attention heads copy tokens they attend to. The QK circuit attends to plausible next tokens, while the OV circuit increases the probability of outputting the attended token, or similar ones.

Some examples of large entries QK/OV circuit

Source Token	Destination Token	Out Token	Example Skip Tri-grams
" perfect"	" are", " looks", " is", " provides"	" perfect", " super", " absolute", " pure"	" perfect... are perfect", " perfect... looks super"
" large"	" contains", " using", " specify", " contain"	" large", " small", " very", " huge"	" large... using large", " large... contains small"
" two"	" One", "\n ", " has", "\r\n ", "One"	" two", " three", " four", " five", " one"	" two... One two", " two... has three"
"lambda"	" $\\", "}{\\", "+\\", "(\\", " ${\\"	"lambda", "sorted", " lambda", "operator"	"lambda... $\\lambda", "lambda... +\\lambda"
"nbsp"	"&", " \"&", "}&", ">&", "=&"	"nbsp", "01", "gt", "00012", "nbs", "quot"	"nbsp... ", "nbsp... > "
"Great"	"The", " The", " the", " contains", " /"	" Great", " great", " poor", " Every"	"Great... The Great", "Great... the great"

Figure 15: Examples of copying in one-layer transformers. Observing the large entries in the QK and OV matrices reveal that some attention heads get specialized in copying previous tokens. Here, the source token is fixed. Destination tokens were obtained by looking at the largest corresponding entries in the QK matrix. Out tokens were obtained by looking at the largest corresponding entries in OV matrix. For example, when the tokens "perfect" and "are" appear in the context, in certain heads, "are" attend strongly to "perfect" and increase the probability that "perfect", or similar tokens such as "super", will be predicted after "are". These heads are somewhat copying previously encountered tokens [2].

Advanced: Eigendecomposition of OV and QK circuits

To identify copying behavior, the authors recommend examining the eigendecomposition of the OV and QK circuits. Eigenvalue analysis is particularly useful when studying functions that map from one space to the same space. In this context, the OV circuit $W_U W_{OV}^h W_E$ maps from the token space to itself.

v is an eigenvector of $M = W_U W_{OV}^h W_E$ with an eigenvalue λ if $Mv = \lambda v$. Simply put, an eigenvector of W is a vector that only scales when multiplied by W.
Positive eigenvalues indicate copying behavior (there is a linear combination of tokens which increases the linear combination of logits of those same tokens). Negative eigenvalues, on the contrary, indicate that there is a linear combination of tokens which decreases the probability of those same token being outputted. Imaginary eigenvalues are less straightforward to interpret, in this case some tokens affect the probability of different tokens being the output.

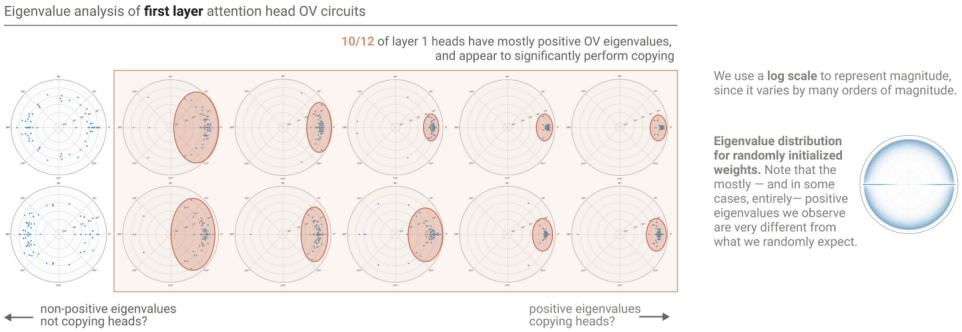

Figure 16: Eigenvalue analysis of first-layer attention head OV circuits. Since eigenvalues are complex numbers, they can be depicted in a 2D representation. Each disk corresponds to a single attention head, with the associated eigenvalues denoted as blue points. 10 out of 12 attention heads appear to mainly have positive eigenvalues, suggesting that they essentially do copying [2].

QK circuits eigenvalues can also be interpreted using eigenvalue decomposition:

- Positive eigenvalues mean that tokens prefer to attend to earlier copies of themselves,
- Negative eigenvalues mean that tokens avoid attending to earlier copies of themselves,
- Imaginary eigenvalues mean that tokens attend to unrelated tokens.

In brief, one layer attention-only transformers can be regarded as a combination of bigram and "skip-trigram" models. While bigrams focus on pairs of consecutive words, skip-trigrams involve examining triplets of words that allow for skipping one or more words in between (sequence of the form `[A]...[B][C]`, examples are given on Figure 15). The bigram and skip-trigram tables can be accessed directly from the weights, using eigenvalue analysis, for example.

Two-Layers Transformers

Compared to one-layer transformers, two-layers transformers can combine attention heads from the first and second layers. Recall that attention heads move information from the residual stream of one token to the residual stream of another; said differently, they move information between tokens.

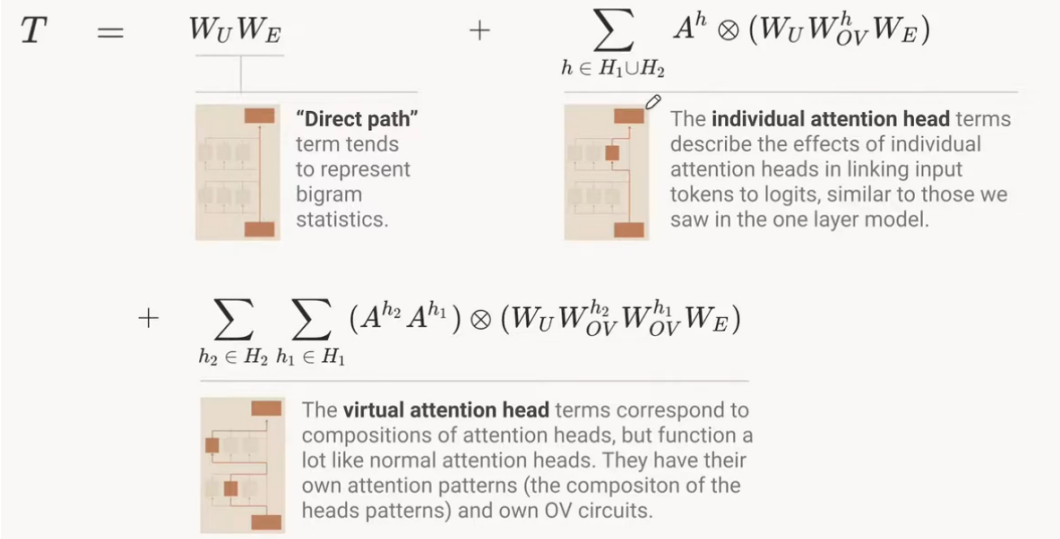

Figure 17: Decomposition of a two-layer transformer into distinct paths. Understanding the tensor product is beyond the scope of this chapter. The key takeaway is to understand that the function the two-layers transformer performs can be broken down into three types of terms: the direct path (that encodes bigrams [A][B] like "Barack Obama"), individual attention head terms (that encode skip-trigram of the form [A]...[B][C]), and virtual attention head terms, that encode more complicated compositional patterns. One-layer transformers could also be broken down into two terms: the direct path and one individual attention head term [2].

Combining attention heads can lead to the emergence of new circuits. For instance, the authors identified a specific type of circuit referred to as **induction heads**. Induction heads consist of two attention heads that predict sequences in the form: `[A][B]...[A]`→`[B]`. Induction heads search for a prior occurrence of the current token `A` within the context, then examine the subsequent token, `B`, and enhance the probability of outputting `B`. They are called induction heads because they perform a form of inductive reasoning (deriving conclusions from a set of observations).

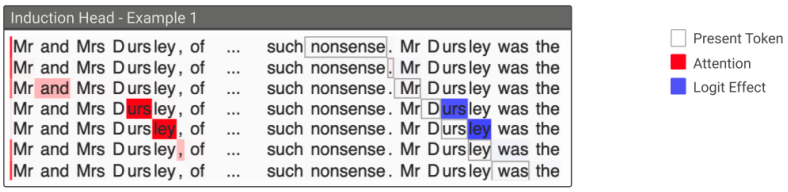

Figure 18: An example of raw attention pattern and logit effect for an induction head. On line 4, for instance, the present token is `D`. It strongly attends to `urs` (in red), which is the next token after the previous instance of `D` in the context. Overall, the induction head increases the probability that `urs` will be repeated [2].

Composition of attention heads across multiple layers enables the emergence of various other attention patterns: there exist heads that attend to the start of the current clause, to the subject of the current sentence, to previous pronouns to determine whether the text is in first, second or third person, to tense markers, etc.

Recap

- Zero-layer transformers model bigram statistics.
- One layer attention-only transformers are an ensemble of bigram and "skip-trigram" models.
- Two layer attention-only transformers can implement more complex algorithms using compositions of attention heads, such as induction heads. Those are further studied in the following paper of the *Transformer Circuits* series: *In-context learning and induction heads* [24].

6 In-Context Learning and Induction Heads

In-Context Learning and Induction Heads [24] is the second paper in the *Transformer-Circuits Thread*. It directly follows *A Mathematical Framework for Transformer Circuits* [2]. In this paper, Olsson et al. delve deeper into the **induction head** circuits previously identified in *Mathematical Framework*. They argue that these circuits are the primary mechanism driving a capability exhibited by LLMs, known as **in-context learning**. The main evidence supporting their hypothesis comes from the observation that in-context learning is acquired at the same time as induction heads form during training, through a phenomenon called a **phase change**.

Before delving into the paper itself, let's first explain what induction heads, in-context learning, and phase changes are.

6.1 Induction Head Circuit

Induction heads are a type of circuits that various kinds of transformers appear to develop fairly early in their training process. Induction heads predict that previous completions will occur again. Imagine that tokens `A` and `B` appeared successively in the context previously. When `A` occurs again, the induction heads predict that `B` will follow `A` once more. **Induction heads predict sequences of the form : `[A][B]...[A]`→`[B]`.**

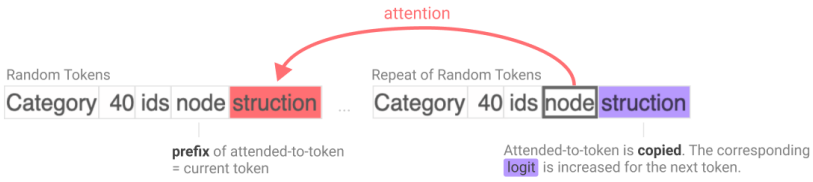

Figure 19: The predicted next token is highlighted in purple, and the current token is `node`. The induction head attends to the token right after (in red) the last occurrence of the current token, and increases the likelihood of `struction` given `node struction...node` [24].

Let `A` be the current token, the induction head looks back over the context preceding it for previous instances of `A`. It then identifies the succeeding token, say `B`, and predicts that `B` will be repeated [11] ([A][B]...[A]→[B]).

An induction head is formed of two attention heads, belonging to two distinct layers, which makes it impossible for one-layer transformers to acquire them (because single layer only encore skip-trigram of the form [A]...[B]→[C]).

Induction heads are much more important than what they look, as they can perform more sophisticated behaviors than mere copy. They sometimes promote completions of the form `[A*][B*]...[A]`→`[B]`, where `A*`≈`A` and `B*`≈`B` are similar in some embedding space. For instance, `B` and `B*` could represent the same word in two distinct languages, or they might share similar meanings.

[11] Induction heads are named by analogy to inductive reasoning.

Inductions heads illustrated

The clearest explanation of the induction mechanism to date is from Callum McDougall [15], who has illustrated the induction head mechanism with clear diagrams. The mechanism is not that easy to implement, and requires decomposing the vector space in which the vectors are embedded into subspaces, with each subspace containing different types of information about the tokens: For example, one of the vector subspaces contains information about the position of the tokens, another vector subspace may contain information on "what is the token", another subspace can say "the next token will be", etc... The picture below explains how a model that has not been trained on Harry Potter and does not know the name "D|urs|ley", can predict the token "urs" following the token "D" after seeing the tokens "D|urs" upstream in the context window. There are two types of compositions: K-composition and Q-composition. The picture below is an explanation of K-composition.

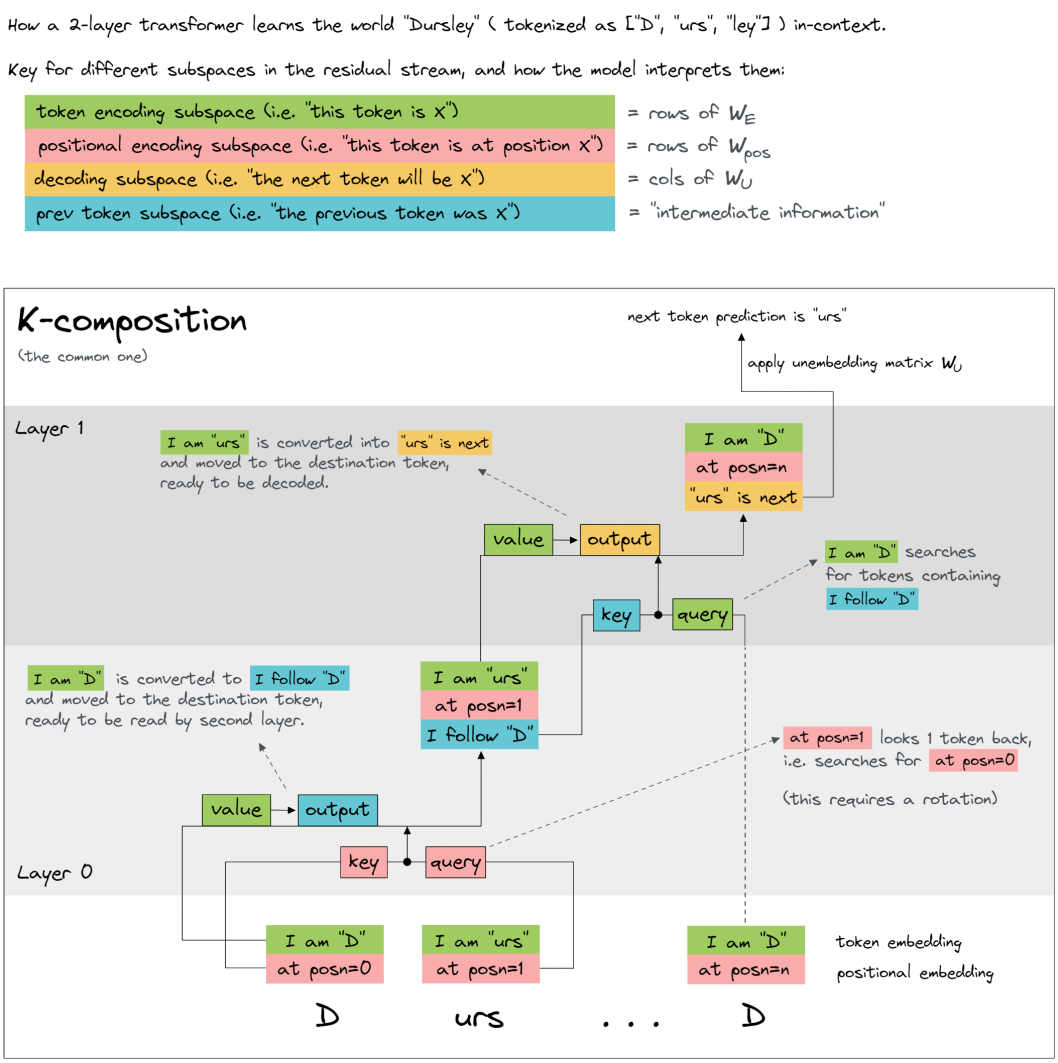

Figure 20: The first step of the K-composition is for each token to look at which token preceded it. The second layer can then search back for the token following the first token "D" (which is "urs") and uses that as a prediction [15].

6.2 In-context Learning

In-context learning is the ability of LMs to use information from far back in the context to predict the next token. Empirically, LMs are better at predicting tokens located later in the context compared to earlier ones (this is quite intuitive, as longer contexts provide more elements for

precise prediction). This ability for in-context learning is significantly more advanced in transformers compared to other language model architectures, such as LSTMs. As said previously, Olsson et al. argue that the in-context learning abilities of transformers hinges on induction heads.

The in-context learning ability cannot be measured directly. Therefore, to observe how it evolves across training, the authors came up with a handcrafted metric for it: (loss of 500th token) - (loss of 50th token), averaged across a large number of sequences[12]. It is denoted in the figures below as the *In-Context Learning Score*.

In-context learning as few-shot learning

The aforementioned definition of in-context learning, decreasing loss at increasing context index, represents one conception of in-context learning.
A second conception associates in-context learning with **few-shot learning**, which is a model's ability to acquire new capabilities during inference, without requiring neither re-training nor fine-tuning. In-context learning challenges the widespread notion that sees a model's behaviors and capabilities as being fixed after training.

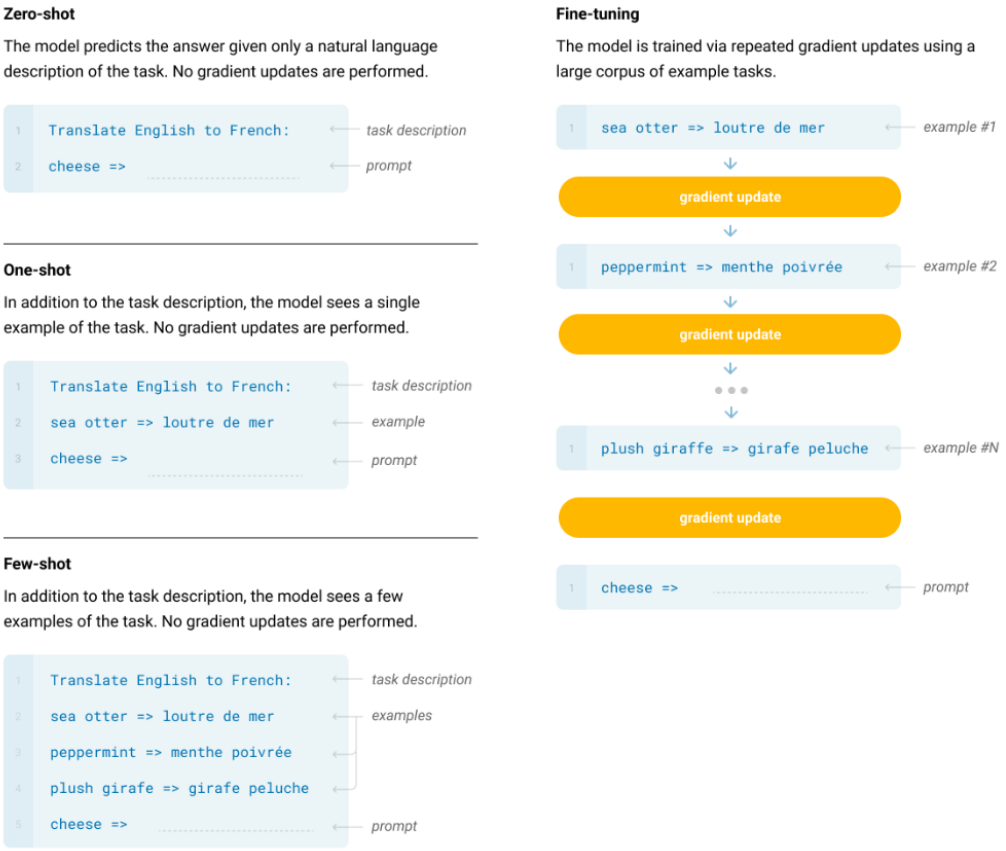

Figure 21: Examples of in-context learning (left: zero-shot, one-shot and few-shot learning), constrasted with traditional fine-tuning(right). In the scenario of in-context learning, the model is given a task description, a prompt, and examples that demonstrate the task. It then learns the task directly from the context (meaning with only forward passes during testing, without any weight updates) [8].

In-context learning is a significant challenge for AI safety because it complicates our ability to predict how models will behave across extended contexts.

[12]Olsson et al. acknowledge that this metric is rather arbitrary, but slight modifications to it do not significantly affect the observations.

6.3 In-Context Learning is Acquired Through a Phase Change

Phase changes (also called phase transitions) happen when a model quickly develops novel abilities or behaviors as it is scaled up or trained longer [20].

The core claim of the paper is that transformers seem to learn fundamental circuits called induction heads, which are responsible for the a significant part of in-context learning in transformer models. During training, transformers undergo a phase change, where the ability for in-context learning is acquired and induction heads form.

What makes the authors confident in the fact that induction heads might be mainly responsible for in-context learning is the co-occurrence of several phenomena within the same timeframe during training:

- In-context learning ability suddenly improves (Fig 22 and Fig 23),
- Induction heads form,
- Model loss has a visible bump, indicating that models may suddenly learn a new important capability at the same time (Fig 24),
- A sudden change in models' performance is also observed in the space of "per-token loss analysis" that we will describe later on (Fig 26)

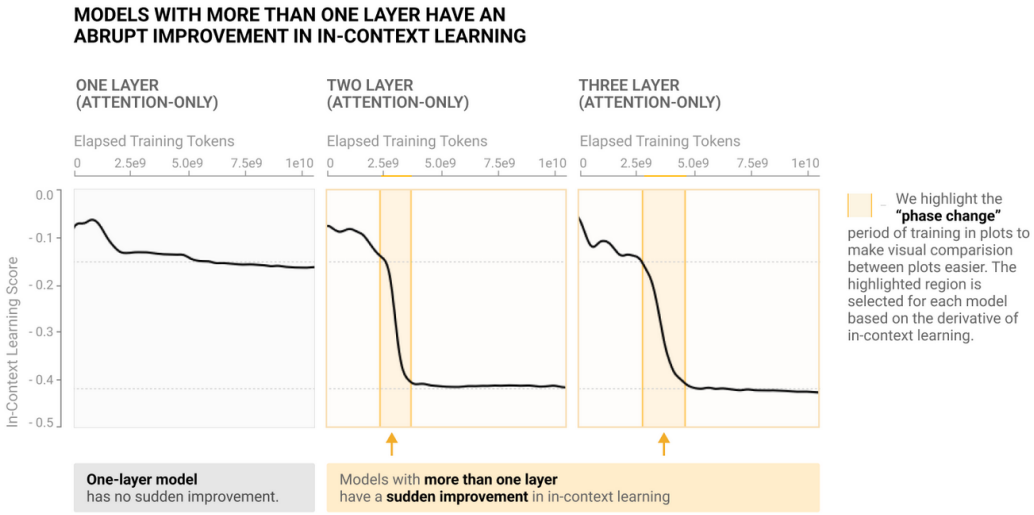

Figure 22: Evolution of in-context learning score across different models and training steps (defined as the 50th token loss minus the 500th token loss). One layer models do not acquire in-context learning abilities, while two and three layers have a sharp transition in score: this is how the phase change was defined. The drop in score indicates that two-layer and three-layer models suddenly get much better at predicting tokens later in the context compared with earlier ones. The score then remains constant for the rest of training. These tendencies hold true across various models' architectures and sizes [24].

Figure 22 helps to get a first intuition about what a phase change is, but the in-context learning score is somewhat arbitrarily defined, raising concerns that the observed phase transition could potentially be a mere artifact. To confirm the validity of the observed phase transition as a robust phenomenon, Olsson et al. examine the derivative of the loss with respect to the token index. This approach allows them to observe how the model's performance improves with regards to the context length.

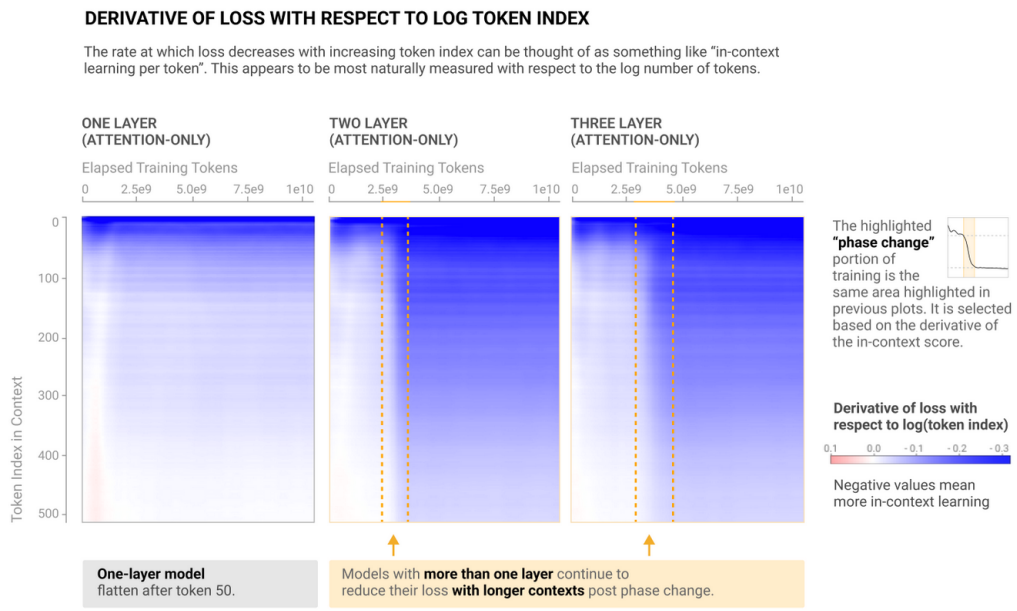

Figure 23: Prior to the phase transition, the loss derivative remains close to 0: the n-th token and the (n+1)-th token are predicted with nearly identical losses, indicating the absence of in-context learning. However, during the phase transition, the loss derivative undergoes a sudden decrease, signifying that the (n+1)-th token is predicted more accurately than the n-th token (= there is in-context learning!) [24]

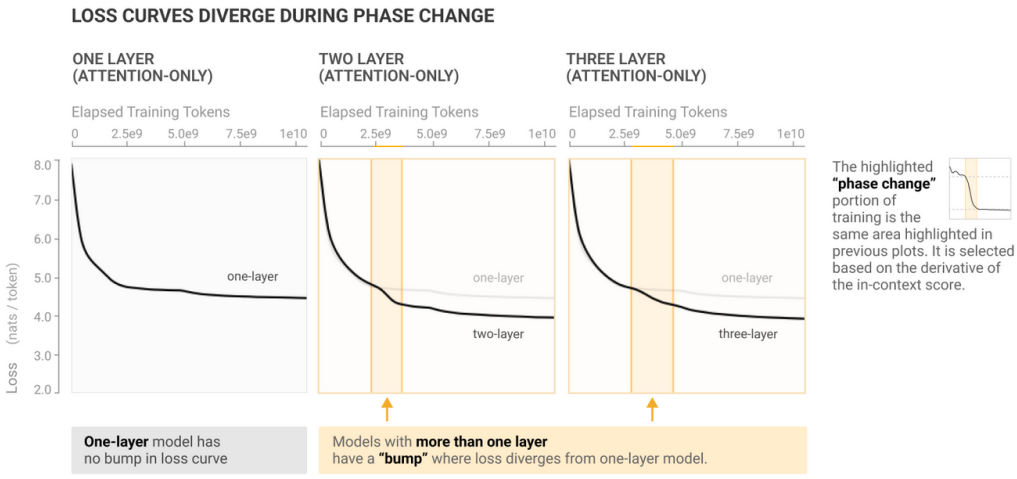

Figure 24: Here is another clue suggesting that "something happens" during the phase change. The important thing to note and retain from this figure is the bump in the loss curve that occurs during the phase change in two and three-layer models (remember that one-layer models can't form induction heads). Interestingly, the loss curves of the three types of models perfectly match each other before the phase transition takes place [24].

The conclusion suggested by the authors based on these observations is that **induction heads are formed during a phase transition. Within the same timeframe, in-context learning abilities dramatically improve**. The co-occurrence of these two phenomena suggests a causal connection between them.

Emerging capabilities during phase transition

Phase transitions are a broad concern in AI safety because they indicate that various capabilities could potentially emerge very quickly during training. As for now, predicting when models will start to show specific skills or become capable of specific tasks is impossible.
Various examples of emerging capabilities during phase transition are documented in the literature:

- GPT-3 is considered the first "modern LLM" in part because of its capacity for *few-shot learning* and *chain-of-thought reasoning*. These abilities were discovered post-training and deployment and emerge unpredictably as a byproduct of increasing investment [7],
- Multi-digit addition is a capability unique to GPT-3 that previous models lacked. This skill is acquired through a phase transition during training [20],
- Pan et al. provide an example of a phase transition in reinforcement learning agents, where better models start following completely different strategies [25]

6.4 "Per-token loss analysis" Method

Olsson et al. studied induction heads and in-context learning across many different models with various architectures, sizes, and at different training stages. To visualize the important trends in models' performance as they vary the parameters, they propose a method they call the "per-token loss analysis". This method involves the following steps:

- **Loss Collection**: Each model is run through a set of data examples. For each example, the loss associated with a specific token is recorded.
- **Loss Vector Creation**: The losses from a specific token across all examples are compiled into a vector.
- **Dimensionality Reduction**: To compare models in a simple and visual manner, these vectors of per-token losses are then processed with principal component analysis (PCA).

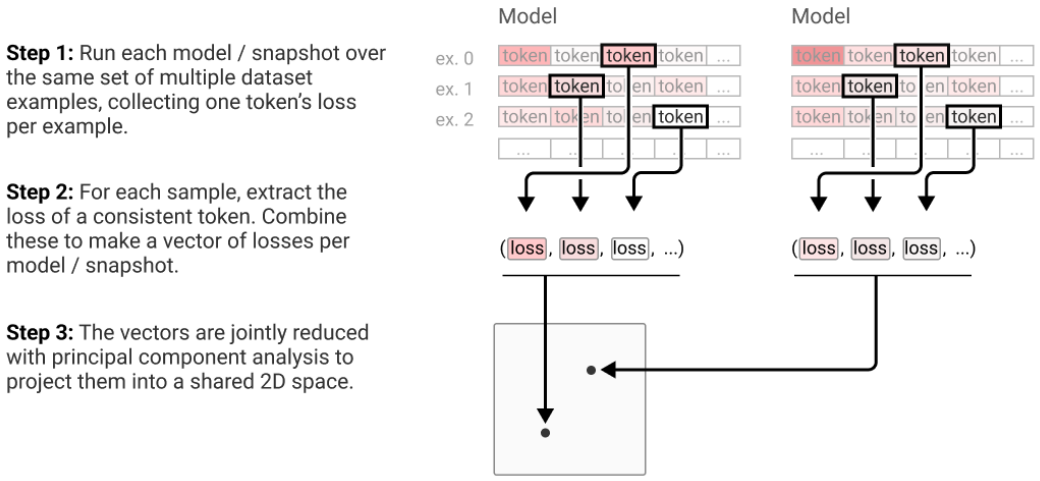

Figure 25: The "Per-token loss analysis" method illustrated. The vector of per-token losses is a way to map different network behavior to the same vector space. In the end, each investigated model is represented by a dot in a 2D space defined by the two first dimensions extracted from PCA, tracing out a trajectory in function space over the course of its training [24].

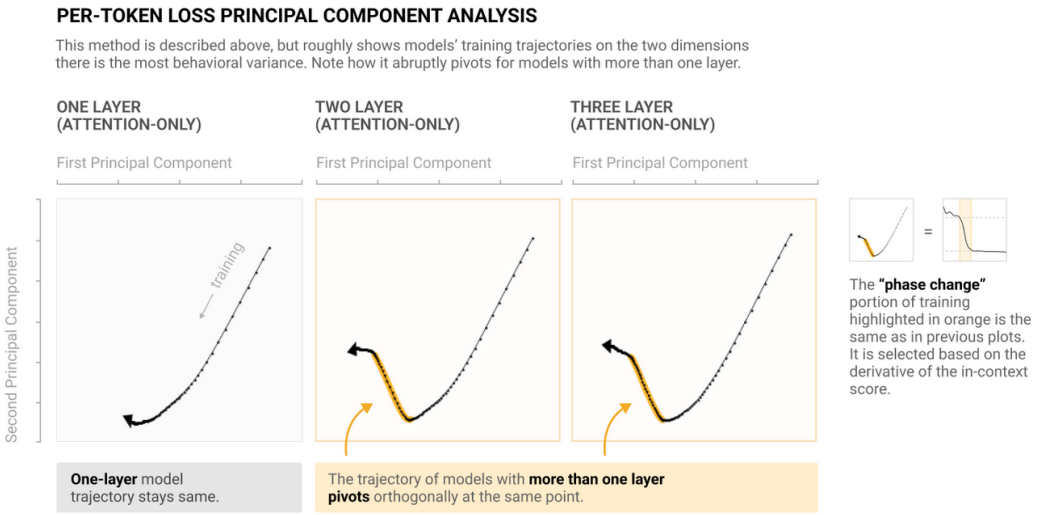

Figure 26: The "per-token loss analysis" method enables to observe how models' improve over training, while taking into account various model types. When the phase change occurs, the model's path in the projected space of per-token losses abruptly shifts direction [24].

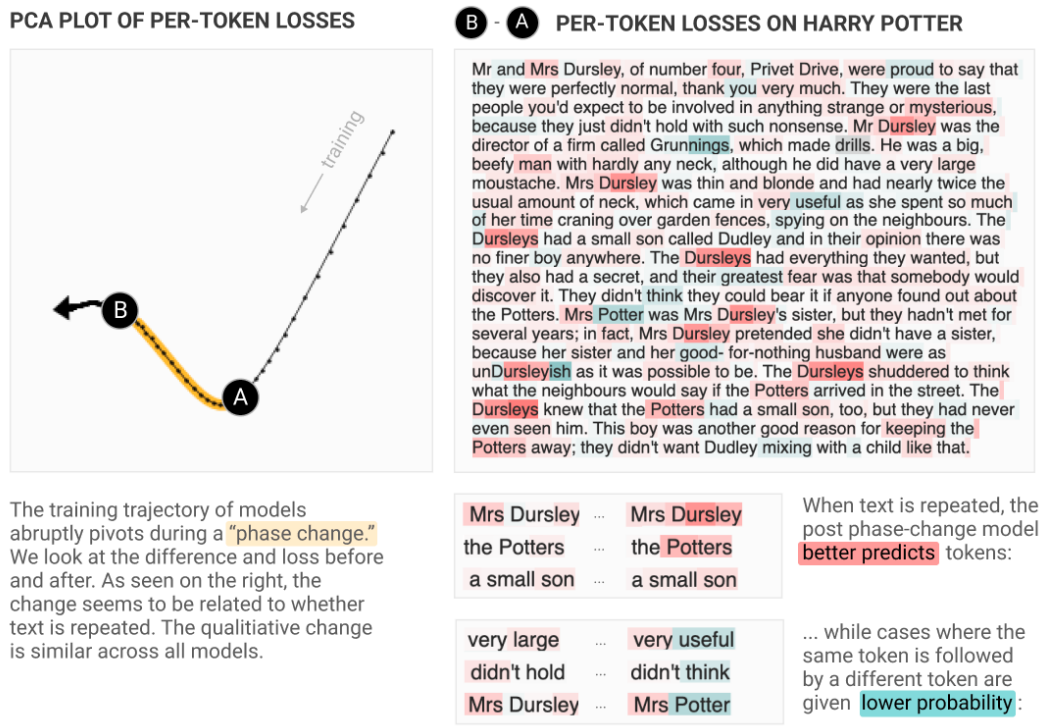

Figure 27: Taking a closer look at how the model's trajectories shift during the phase transition using an example helps us gain a clearer understanding of the process. Point B on the graph represents the end of the phase transition, while point A corresponds to the start. The color scale shows which of the two models predicts each token more accurately. The model after the phase transition is better at predicting tokens that appeared previously in the text (meaning it's better at in-context learning). However, the loss for the token "Potter" increases after the phase transition because in-context learning should predict "D" for "Dursleys" [24].

6.5 Recap

Transformers seem to learn fundamental circuits called induction heads, which are responsible for the majority of in-context learning in transformer models.

During training, transformers undergo a phase change, where the ability for in-context learning is acquired and induction heads form.

The authors present several supporting arguments suggesting that induction heads might account for part of in-context learning. These arguments include:

1. As seen already, induction heads are formed during a phase transition. Within the same timeframe, in-context learning abilities dramatically improve. The co-occurrence of these two phenomena suggests a causal connection between them.

2. The first argument is based on observations. To strengthen the validity of their hypothesis, Olsson et al. later conduct a more "interventional" experiment where they perturb the transformer architecture to demonstrate the causal relationship between induction heads and in-context learning. By altering the architecture to facilitate the formation of induction heads, they observe a corresponding change in in-context learning capabilities.

3. A third experiment they conducted involved ablation, where deleting an attention head during testing leads to a decrease in in-context learning abilities.

Event though the authors mention that the evidence is not as compelling for large models as for small ones, the *Transformer-Circuits Thread* demonstrates that the analysis of toy models can effectively reveal fundamental functions of deep learning models.

To delve deeper and provide further clarification, it is recommended to watch the walkthrough video in which Olsson presents the paper's key findings.

7 Designing More Interpretable Model Architectures: Softmax Linear Units (SoLU)

The paper *Softmax Linear Units* [11] builds upon the foundational work of two previous papers in the *Transformer-Circuits Thread*. The first paper laid the groundwork by reverse-engineering simple attention-only transformers, while the second provided evidence for the role of induction heads in in-context learning. *Softmax Linear Units* extends this exploration to MLP neurons, which are challenging to interpret due to polysemanticity — they respond to multiple unrelated features. The authors propose **SoLU**, an activation function aiming to make neurons less polysemantic, thus more interpretable. Their results suggest SoLU increases the fraction of interpretable neurons and maintains competitive performance.

Prior to delving into the paper, it's important to establish a clear understanding of what are features, polysemanticity, the superposition hypothesis and privileged bases in neural networks.

7.1 Background on Mechanistic Interpretability

Within a transformer MLP, a **neuron** refers to a single activation. One can define a standard basis over the activation space, where each canonical direction corresponds to a single neuron's activation.

A **feature** refers to a human-understandable pattern in the activation of a network, or a direction in the activation space. It is possible that a neuron strongly activates only in presence of a single specific feature, in which case it's pure. Often, neurons respond to multiple apparently unrelated features together. These are called **polysemantic neurons**. They can be found in different types of architectures; for instance, InceptionV1 contains a neuron that responds strongly to both images of cats and cars. These polysemantic neurons were initially detected in vision models through feature visualization and pose a significant challenge for interpretability because they do not cleanly map to individual and easily identifiable concepts [23].

The superposition hypothesis

The **superposition hypothesis** was proposed to explain why polysemantic neurons emerge in neural networks. It posits that models have more features to learn than they have neurons, making it impossible for each feature to have a dedicated neuron. The most efficient way to represent more features than there are neurons is for features to share neurons. Otherwise, having a single neuron correspond to a single feature would result in a wasteful allocation of parameters. **Polysemanticity is expected if features do not align with the privileged basis of neuron activations**. But networks that have features aligned with their neurons are much more interpretable.

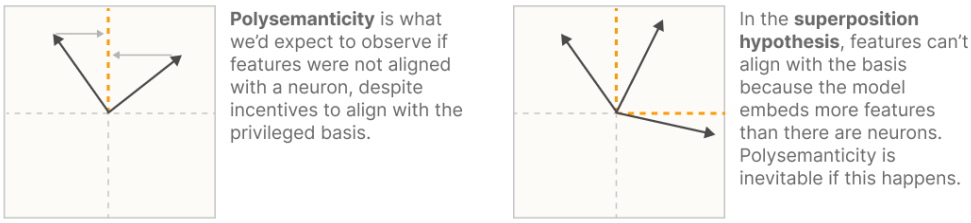

Figure 28: Relationship between polysemantic neurons and the superposition hypothesis. Polysemantic neurons emerge as a means to represent a greater number of features, albeit with the drawback of introducing 'interference' between them [11].

An important concept to grasp is the notion of a **privileged basis**. Normally, in linear algebra, each dimension or coordinate in a space is treated as having equal "importance". There's no inherent reason to consider any particular coordinate as more special or significant than others. This is the concept of "non-privileged basis". To get an intuition about what a privileged basis is, consider a layer using

the ReLU activation function. By setting all negative activations to zero, ReLU causes the network to favor positive activation values because negative ones do not transmit information to the following layer. Thus, to minimize feature interference and maximize the number of encoded features, the most effective arrangement is for features to align along the privileged basis. When a layer in a neural network uses ReLU, features should tend to align with the standard basis axes, making this basis a privileged one[13].

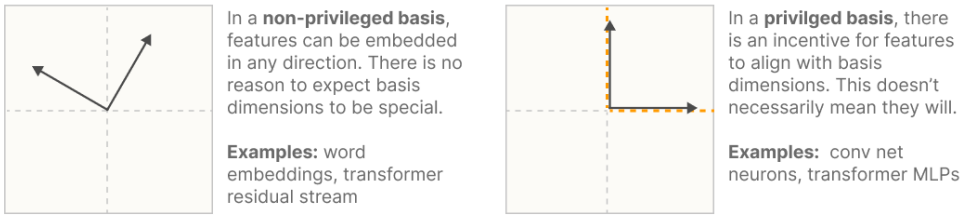

Figure 29: Difference between a privileged and a non-privileged bases in neural network activation space. In a non-privileged basis, there is no incentive for features to align with the directions of neurons (also known as standard basis). Examples include transformers token embeddings, residual streams, and attention vectors. On the other hand, MLP activations are privileged because MLPs contain a non-linear activation function. Thus, features should tend to align with neurons. In absence of a privileged basis, looking at the neurons with the hope of interpreting them doesn't really make sense [11].

7.2 SoLU Increases Interpretability with Limited Performance Cost

The SoLU activation function is designed to encourage models to acquire features that align with neurons. It effectively **increases the fraction of MLP neurons which appear to have clear interpretations, all while preserving performance**.

The SoLU activation function is defined as:

$$SoLU(x) = x * softmax(x)$$

It has interesting properties aimed at reducing polysemanticity by incentivizing features to align with the privileged basis:

- When SoLU is applied to a vector of large and small values, the large values will suppress smaller values: $SoLU(4, 1, 4, 1) \approx (2, 0, 2, 0)$
- Large basis aligned vectors are preserved: $SoLU(4, 0, 0, 0) \approx (4, 0, 0, 0)$
- Feature spread across many dimensions will be suppressed to a smaller magnitude: $SoLU(1, 1, 1, 1) \approx (\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4})$

To assess whether the new architecture is more interpretable or not, a group of evaluators were tasked with categorizing neurons as interpretable or non-interpretable, based on a sample of sentence that elicit the strongest activation. Across models of 1 to 40 layers, SoLU enhances the fraction of interpretable neurons by approximately 25%.

[13]Note however that even if a features are perfectly aligned with the standard basis, one canonical vector in the standard basis can still represent two very distinct features if the neural network does not find it necessary to separate them.

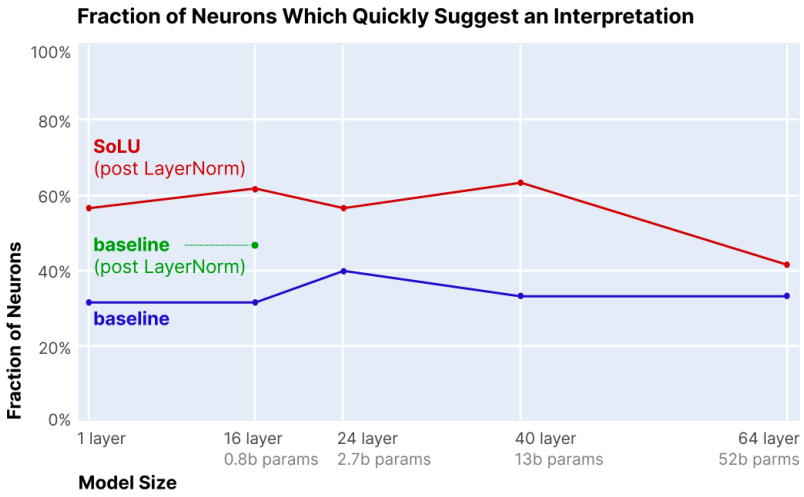

Figure 30: SoLU seems to enhance interpretability. The colored lines represent the proportion of neurons classified as interpretable for each model type. The baseline transformer employs the traditional GeLU activation function [11].

7.3 What Features do Transformers Learn ?

Similar to the approach taken in the *Thread: Circuits* [23], Elhage et al. try to understand how transformers represent information by looking at individual neurons throughout the model.

- **Early layer neurons**: Elhage et al. observed neurons responding to multi-token words (e.g., 'not|withstanding,' 'word|ing,' 'civil|ian') or compound words (e.g., 'International|Monetary|Fund,' 'George|Orwell,' 'computer|vision,' 'heart|attack'). They also identified distinct neurons responding to the same token but in different contextual settings. For instance, three different neurons respond to the word 'die,' each triggered when it's used in texts in Dutch, German, or Afrikaans. Many neurons of the form 'token X in language Y' appear to emerge. Additionally, some neurons respond differently to the '<' character based on the programming language in which it's employed.
- **Middle layer neurons**: These neurons represent more intricate or abstract concepts. Examples include neurons responding to numbers only when they refer to a specific number of people, neurons activated by discourse markers emphasizing the importance of a statement (e.g., 'the amazing thing is'), and neurons that disambiguate the special interpretation of a token, such as a neuron responding to grades represented as A/B/C/D. There's even a neuron seemingly aiding in the parsing of columns in ASCII tables.
- **Late layer neurons**: late layers often exhibit a complementary function to early layers. While early layers recognize and 'de-tokenize' multi-token words, late layers occasionally 're-tokenize' compound words back into tokens.

7.4 Limitations and Further Directions

It's worth noting that SoLU does not make all neurons easily interpretable, but even if one were to successfully explain the function of every neuron, it would still be insufficient. Developing methods for creating a more concise representation or summarization, similar to the approaches discussed in [22], is needed. Moreover, the benefits of SoLU on interpretability appear to decline significantly as models increase in size. Elhage et al. also mention that while SoLU offers advantages, it may come at the expense of 'masking' other features.

8 Towards Monosemanticity: Decomposing Language Models With Dictionary Learning

In the paper titled *Towards Monosemanticity: Decomposing Language Models With Dictionary Learning* the authors finally tackle MLP layers [34]. Because MLP neurons are highly polysemantic (they respond to a variety of stimuli and can't be associated with a single, distinct concept), they can't be straigthforwardly interpreted. Polysemanticity arises because MLP layers represent more features than they have neurons.

To extract the underlying features present in MLPs, the authors suggest training an autoencoder to reconstruct MLPs activations. To achieve this, the autoencoder must be *overcomplete*, meaning that its latent space is larger than its input. Through training, the neurons in the autoencoder's latent space are expected to represent disentangled and interpretable features. This method is referred to as "dictionary learning."

In SoLU [11], the model is encouraged to create monosemantic features during initial training, rather than disentangling them afterward. However, the authors of *Towards Monosemanticity* ultimately conclude that this approach is not viable.

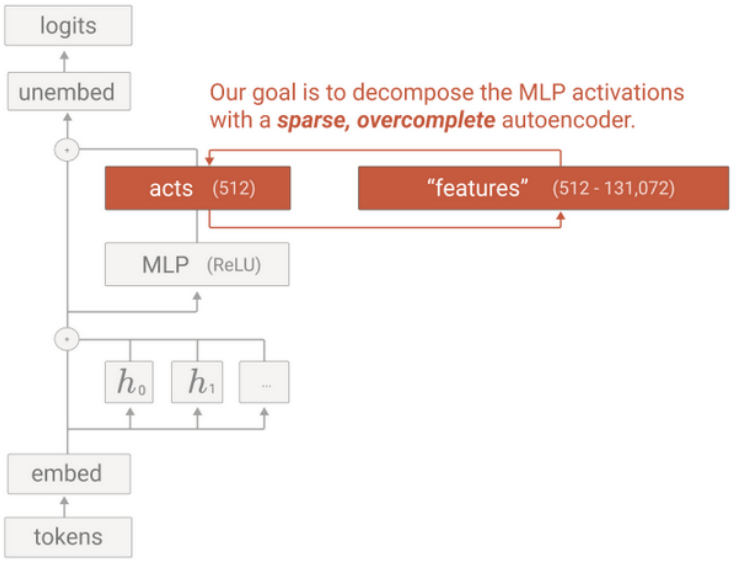

Figure 31: MLP activations can be decomposed into features using an autoencoder [34].

Transformer MLP layer

Tranformers MLP layers have one hidden layer (typically 4 times larger than the residual stream). Here is what an MLP m is computing:

1. Reads in the residual stream a tensor x_i $[d_{model}, n_{context}]$,
2. Multiplies it by its input weights W_I^m $[d_{mlp}, d_{model}]$,
3. Applies an activation function (typically GeLU or ReLU),
4. Projects the result back into the residual stream through its output weights W_O^m $[d_{model}, d_{mlp}]$,
5. Adds the result to the residual stream.

$$x_{i+1} = x_i + W_O^m ReLU(W_I^m x_i)$$

8.1 Setup

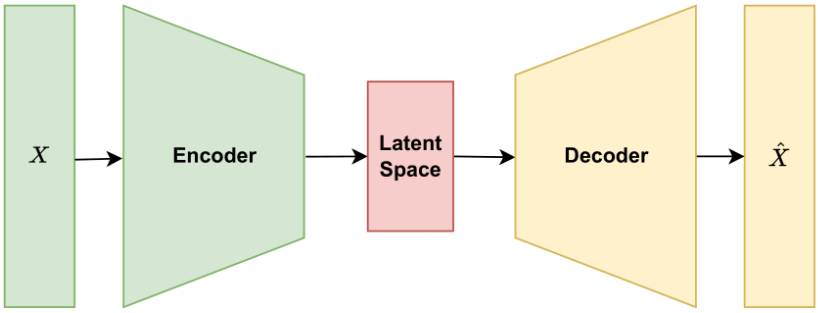

Figure 32: Autoencoder. Let d_{mlp} be the input and output dimension, m the dimension of the latent space. An autoencoder learns two linear transformations: represented by encoder weights W_e $[m, d_{mlp}]$ and decoder weights W_d $[d_{mlp}, m]$, and biaises b_e $[m]$ and b_d $[d_{mlp}]$. **Encoding:** $f = ReLU(W_e(x - b_d) + b_e)$. **Decoding:** $\hat{x} = W_d f + b_d$. Feature activations are the output f of the encoder.

The autoencoder has an expansion parameter, ranging from 1 to 256, that controls the latent space dimension ($m = \exp d_{mlp}$). The bigger m is, the more features can be extracted.

It was trained on 8B datapoints. While the dictionary learning approach had been explored in prior research, the authors highlight that, in this study, they employed significantly more computational resources for training the autoencoder compared to previous attempts.

The authors focus on one-layer transformers, with 12 attention heads, and an MLP layer consisting of 512 neurons with a ReLU activation function. Remarkably, even with just 512 neurons (but with a high expansion parameter), they were able to discover tens of thousands of distinct features in the autoencoder.

8.2 Results

Key results:

- Sparse autoencoders extract relatively monosemantic features.
- These features can't be directly found in the neuron basis.
- Features can be used to steer transformers.
- Learned features seem to be quite universal. Different training runs learn similar features.
- Increasing the size of the autoencoder causes features to split and get more specialized.

The authors discovered a wide range of distinct features that activate for various inputs, including text written in Arabic script, DNA sequences, base64 strings, and more.

Let's delve into the DNA feature in greater detail. It's important to ensure that the learned features exhibit specific characteristics:

- **Activation specificity**: the feature activates specifically on the hypothesized context (the DNA feature activates solely in presence of sequences of `A`, `T`, `C` and `G`. See Fig 33).
- **Activation sensitivity**: in the hypothesized context, the feature activates (when `A`, `T`, `C`, `G` sequences are present, the feature generally activates. See Fig 33).
- **Causal effect**: the feature causes appropriate downstream effects. It is possible that the learned features might only capture surface-level patterns in the data rather than the underlying properties of the model. To make sure that this is not the case, the authors check that the features can mediate downstream effects on the logits (i.e. the features can be used to influence the distribution of logits). The autoencoder has a causal effect if, for instance, when the DNA feature is activated, the model becomes more inclined to produce `A T C G` tokens. If the distribution of logits can be

influenced by acting on the autoencoder latent space, it strongly suggests that the autoencoder explains causal aspects of the MLPs (See Fig 34).

- The feature doesn't correspond to any neuron, because if features can be found in the neuron basis, then the dictionary learning method is useless (See Fig 35).

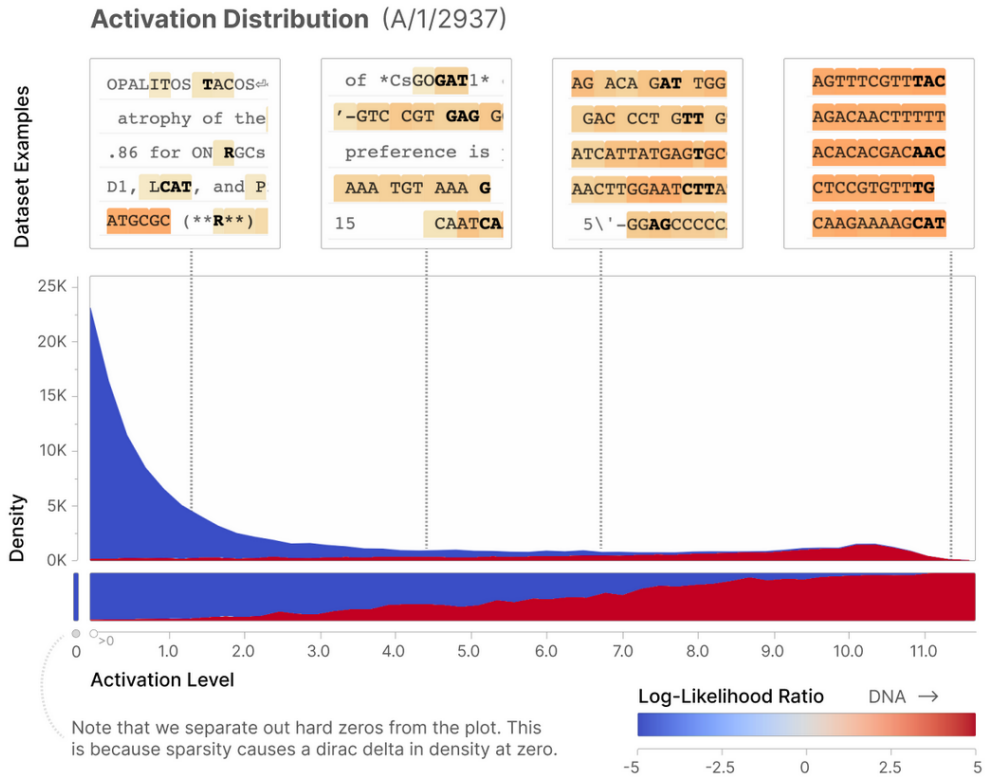

Figure 33: Activation distribution of the DNA feature. The log-likelihood ratio is a proxy for whether a sequence s is a DNA sequence; it is defined by $log(P(s|DNA)/P(s))$. The higher the activation of the DNA feature over a sequence, the more likely the sequence is DNA, which is indicative that the feature is sensitive to DNA sequences and activates specifically to DNA over a certain level [34].

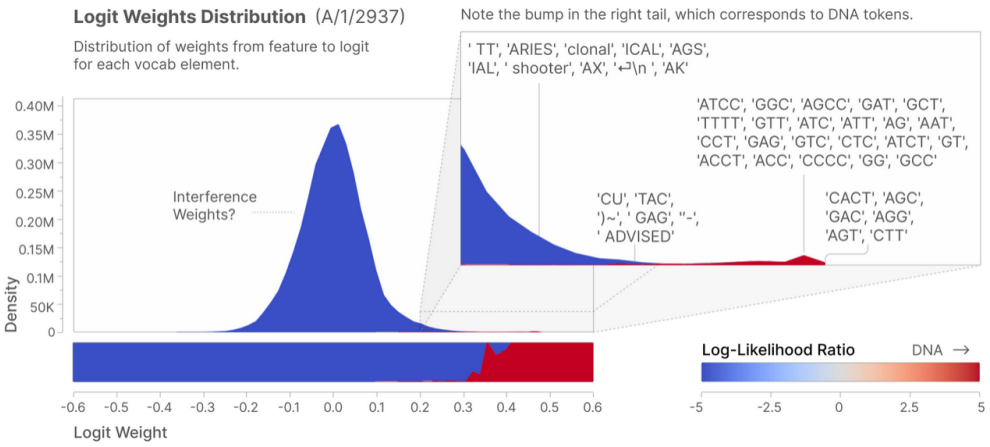

Figure 34: MLPs activations can be approximated as a sum of features f_i and a bias b: $x^j \approx b + \sum_i f_i(x^j)d_i$, where d_i represents the direction of feature i in the MLP activation space (d_i is the i-th column of W_d, the decoder weights). The direction of feature i, d_i, when multiplied by the MLP output weights, then by the transformer unembedding matrix, provides a vector of logit weights. This is how the logit weights distribution is obtained. Tokens that correspond to DNA sequences (the ones that have a high log-likelihood ratio) tend to have high logit weights. This means that **activating the DNA feature increases the probability that the transformer predicts `A T C G` sequences** [34].

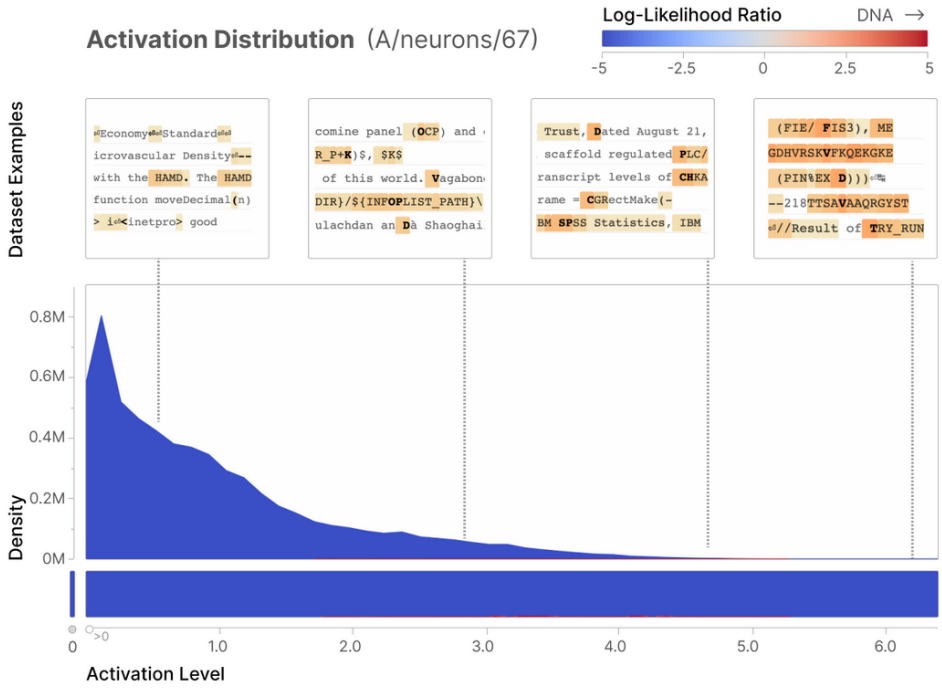

Figure 35: A/neurons/67 refers to the MLP neuron whose activations correlate the most with the DNA feature activations. The distribution of its activations, contrasted with the log-likelihood ratio of DNA, indicates that DNA-related tokens only account for a very small portion of the tokens that highly activate neuron 67 [34].

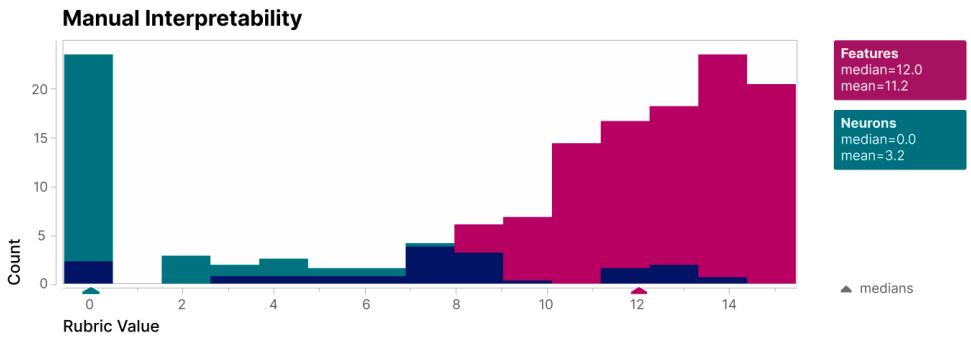

Figure 36: To analyze the interpretability of features compared to neurons, an annotator provided explanations and rated their confidence on a scale from 0 to 15. The results suggest that features are more interpretable than neurons! [34]

The paper *Towards Monosemanticity* explores the use of a sparse autoencoder to extract interpretable and monosemantic features from transformer MLP layers that are often not visible in the neuron basis. These features can also be used to intervene in and steer the model's generation.

9 Reverse-Engineering a Full Circuit in GPT-2 small

Interpretability in the Wild [37] is the first paper that aims at reverse-engineering and end-to-end behavior in a language model. Basically, when given as input: "When Mary and John went to the store, John gave a drink to", GPT-2 small correctly identifies that next token should be "Mary", instead of "John". This completion task is known as *Indirect Object Identification* (IOI), because "John" is the subject of the sentence, and "Mary" the indirect object. Using a variety of interpretability tools and methods, Wang et al. identified specific subparts of the model responsible for this completion.

An overview of the IOI circuit will be presented, then the path patching method that was used to identify it[14]. This paper and the path patching method are particularly complicated to grasp; feel free to skip this part if you feel it's too complex.

9.1 Indirect Object Identification Circuit Overview

The IOI circuit in GPT-2 small consists of various types of attention heads that interact with each other. Fig 37 provides an overview of this circuit.

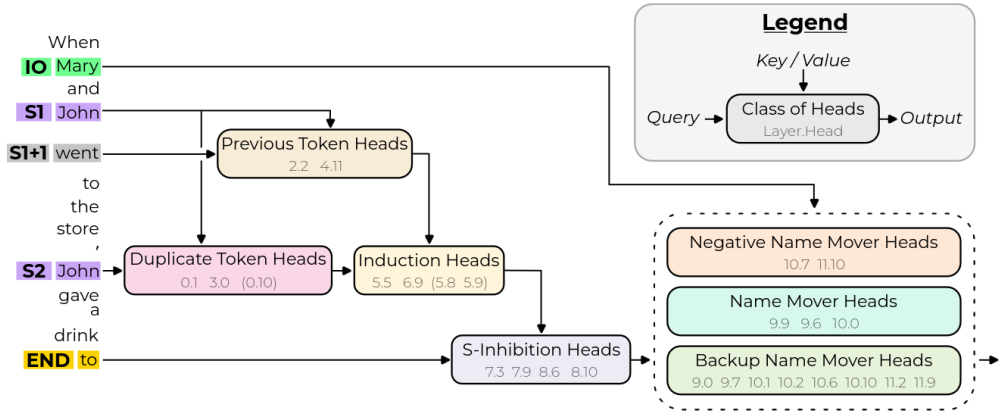

Figure 37: Overview of the IOI circuit in GPT-2 small. Input tokens are listed in a column on the left: "When Mary and John went to the store, John gave a drink to". Pay attention to the location of the query, key/value and output arrows. The key/value arrows specify the residual stream from which the heads read, while the query/output arrows indicate the residual stream to which they write. IO refers to the Indirect Object, S1 denotes the first occurrence of the subject and S2 its second occurrence [37].

Let's start by explaining how this circuit operates, beginning with the deepest heads and progressing back to the input. The IOI circuit comprises three main types of heads. To get a quick first intuition of the circuit, the other ones will be ignored for now.

- **Name Mover Heads** (denoted in a dashed box) attend to all the names within the context (IO, S1, and S2) and copy what they attend to. Figure 37 shows that these heads primarily attend to IO, allowing them to input the correct IO token while ignoring S1 and S2. This is possible because the S-Inhibition Heads, positioned earlier in the network, signal to the Name Mover Heads not to attend to S1 and S2.
- **S-Inhibition Heads** become active at the END token and primarily attend to S2. Their purpose (in this context) is to inform the Name Mover Heads to refrain from paying attention to S2, as it is irrelevant to the IOI task.
- **Duplicate Token Heads** are responsible for identifying previous instances of a token within the context. In this scenario, Duplicate Token Heads become active at the second occurrence of "John" because it has already appeared before. The key/value arrow indicates that these heads primarily attend to the first occurrence of the subject, S1. The reason S-Inhibition Heads can

[14]This was not the only method used, but only this one will be introduced.

inhibit both S1 and S2 while primarily attending at S2 is because Duplicate Token Heads were previously active to signal all occurrences of the subject and their positions. Essentially, Duplicate Token Heads act as pointers to S1.

In brief, the Duplicate Token Heads detect earlier occurrences of the subject, convey their positions to the S-Inhibition Heads, which then inform the Name Mover Heads that S1 and S2 are not likely outputs.

In reality, the circuit is much more complex, and involves induction heads, previous token heads, and other types of name mover heads.

- **Previous Token Heads** transmit information from S1 to the subsequent token, denoted as S1+1.
- **Induction Heads** have a similar role with Duplicate Token Heads, but they employ an induction mechanism. They attend to the token following S1 and signal the S-Inhibition Heads that S1 has been duplicated, effectively acting as a pointer to S1. For a more in-depth explanation of Induction Heads, refer to Section 6.
- **Backup Name Mover Heads** take the role of the name mover heads when they are out of service. They may appear because transformers are trained using dropout.
- **Negative Name Mover Heads** exhibit a rather intriguing behavior. They write in the opposite direction of the names they attend to.

Various methods were explored to identify those heads, among which **path patching**, which enables to measure how one head affects subsequent residual stream vectors and MLPs.

9.2 Path Patching Method

The path patching method helps understanding the indirect impacts of specific model components on the model output or on other components within the model.

Path patching involves **three key steps**:

1. In the initial phase, a forward pass is performed on a sentence like "When Mary and John went to the store, John gave a drink to." The model's activations at each layer are stored.
2. In the second step, another forward pass is conducted on the same sentence, but this time, the three names are randomly set, for example, as in: "When Hannah and Louis went to the store, Paul gave a drink to". Activations are once again stored. In sentences of this type, where two plausible answers exist ("Hannah" and "Louis"), the IOI task lacks sufficient information for an unambiguous completion.
3. To assess **how a specific head h influences the output logits**, a third forward pass is performed using the initial IOI sentence as input. During this pass, all heads are kept frozen from the first forward pass, and only the activations of head h are replaced with those from the second forward pass. This step involves recomputing only the residual stream and MLPs. In essence, if head h doesn't play a crucial role in the IOI task, the patching operation should have minimal impact on the output logits. Conversely, if head h is a vital component of the IOI circuit, differences in logits should become apparent between this third forward pass and the initial unambiguous one. This is because patching the second forward pass removes the necessary information for performing the IOI task.

The described version of path patching is also referred to as **direct logit attribution**. When attempting to reverse-engineer a network, it is often a good strategy to begin by identifying the elements that have the most significant influence on the output and then work backwards.

This is not the only way to do path patching. Instead of considering all the paths from h to the logits, it is possible to restrict the path from h to one or more specific elements within the network. For instance, when seeking to identify the heads that affect the name mover heads, paths of the form $h \rightarrow$ name mover head can be patched.

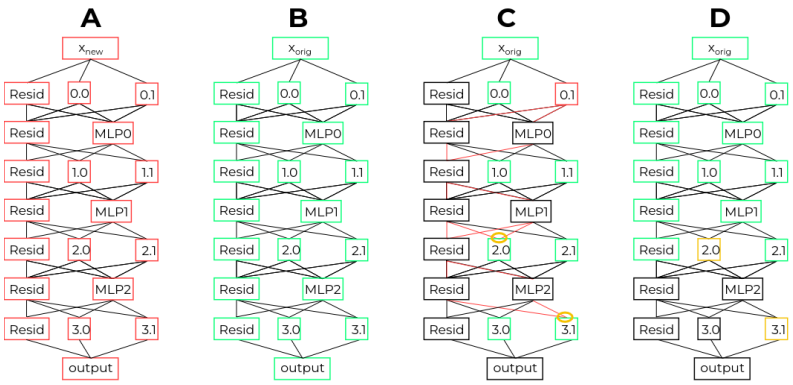

Figure 38: Illustration of the path patching method. x_{new}: "When Hannah and Louis went to the store, Paul gave a drink to", x_{orig}: "When Mary and John went to the store, John gave a drink to". (A) corresponds to step 2 in the previous paragraph, (B) to step 1, (C) and (D) to step 3. However, this figure doesn't correspond exactly to direct logit attribution, rather the aim here is to understand the influence of head 0.1 on both heads 2.0 and 3.1. In this figure, colored heads are either patched or frozen, while black heads are recomputed. In (C), the red paths indicate the paths through which head 0.1 can impact heads 2.0 and 3.1. During this step, a forward pass is conducted with the frozen heads from step 1/(B) and the patched head 0.1 from step 2/(A). The activations of heads 2.0 and 3.1 are frozen, but their activations are retained for the final forward pass in (D). (D) takes the original sentence x_{orig} as input and incorporates the saved activations from the previous step. This last forward pass allows us to measure the effect of head 0.1 on heads 2.0 and 3.1. The indirect impact of head 0.1 on heads 2.0 and 3.1 can be measured by comparing the logits between pass 1 and pass 4 [37].

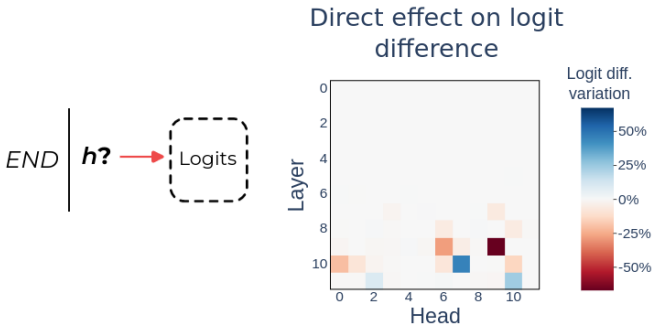

Figure 39: Result of direct logit attribution on the IOI task. The heads that affect the logits the most can be identified. Logit difference measures the difference between the logit for the IO token and the logit for the subject token: $logit(IO) - logit(S)$. A positive logit difference variation indicates that the head, when patched, increases the probability of outputting the IO token. Patching heads 9.6 and 9.9 causes a large drop in logit difference, while patching heads 10.7 and 11.10 causes an important increase. Heads 9.6 and 9.9 are name mover heads; heads 10.7 and 11.10 are negative name mover heads [37].

Now that the heads have the most significant impact on the logit difference were identified, one can examine the preceding heads on which these depend.

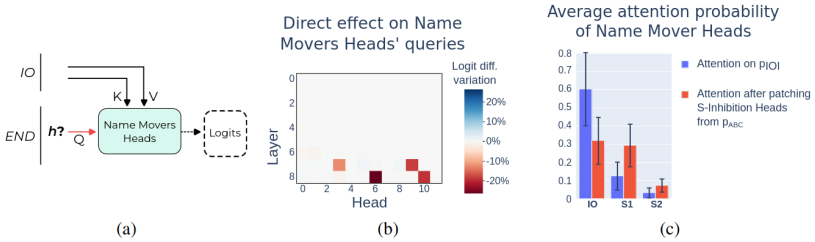

Figure 40: Path patching method used to identify heads that affect the most name mover heads. The paths that were patched are of the form: $h \rightarrow$ name mover heads. (b) Heads 7.3, 7.9, 8.6 and 8.10 cause a drop in logit difference when patched. (c) Path patching can be completed with attention pattern analysis to better understand the roles of these heads. Attention patterns are visualized before patching (blue) and after (red). Patching these heads increases the attention of name mover heads on S1 and decreases it on IO. These heads thus decrease attention on S1, this is why they are called S-inhibition heads [37].

Similarly, one can analyze which heads exert influence on S-inhibition heads and so forth. The authors pinpointed four types of heads (duplicate token heads, induction heads, and previous token heads) and observed their attention patterns to get an intuition of what they do.

Interpretability in the Wild presents an advanced analysis of GPT-2 small's indirect object identification (IOI) mechanism. The authors identify a specific circuit of 26 attention heads essential for IOI. They use causal interventions like path patching and knockouts (a method where specific components (like attention heads) of a neural network are selectively disabled or removed. This technique is used to understand the impact of these components on the model's performance). Key findings include the identification of different classes of attention heads such as Name Mover Heads, S-Inhibition Heads, and Backup Name Mover Heads, each playing a specific role in IOI.

To assess the reliability of their explanation, the authors suggest three main criteria:

- **Faithfulness**: how accurately does the explanation represents the workings of the model ? An explanation is considered faithful if it truly reflects the processes and decisions made by the model.
- **Completeness**: refers to the extent to which the explanation covers the relevant aspects of the model's behavior. A complete explanation should account for all significant factors influencing the model's decisions.
- **Minimality**: evaluates the simplicity of the explanation. The explanation should be as concise as possible.

The authors acknowledge gaps in understanding, but their work represents a significant step towards a mechanistic understanding of transformers.

10 Concept Erasure

Concept erasure is the process of selectively removing specific learned information from a neural network's activations. This is a complex task because it requires finding a trade-off between eliminating a concept as much as possible while causing minimal collateral damage to other concepts (in other words, the operation must be surgical).

The applications of concept erasure are diverse:

- **Improving fairness**: for example, one may want to erase certain concepts that may be considered inappropriate for decision-making, like gender or race, to ensure that a network does not make decisions based on them.
- **Enhancing interpretability**: by observing how a network behaves after specific concepts are removed, one gains insights into networks inner mechanisms and how they processes information.
- **Enhancing robustness**: concept erasure can also be used to enhance a network's robustness by eliminating misleading or spurious learned features.

In *LEACE: Perfect linear concept erasure in closed form*, Belrose et al. introduce a compelling concept erasure method known as LEAst-squares Concept Erasure (LEACE). They demonstrate that LEACE achieves the most precise linear erasure possible[15].

Distinction between features and concepts

First, let's establish an important distinction that will help understanding the explanation coming after:
- A **feature** in this context refers to a pattern in the activation of a network, or a direction in the activation space. Features can range from individual neurons (the smallest features possible) to more complex patterns.
- A **concept** is a human-interpretable variable, such as "gender" or "truth", which can take on different values. In the context of LEACE, a feature is a pattern of activation that is interpreted as encoding a concept.

LEACE makes an important simplifying assumption by focusing solely on removing **linear features** (which means it doesn't address features that might be encoded in nonlinear subspaces within the activation space). This choice is motivated by three key reasons:

- Easier identification: linear features are more straightforward to identify. They just involve finding a linear correlation between activations and a dataset representing a specific concept.
- Reduced overfitting: restricting features to be linear makes it more challenging to overfit to the data used to discover them. This enhances the reliability of the erasure process.
- Orthogonal decomposition theorem: the linear feature assumption enables to use the orthogonal decomposition theorem.

Orthogonal decomposition theorem

The orthogonal decomposition theorem states that any vector within a linear subspace can be uniquely decomposed into two parts: one lying within the subspace itself and the other orthogonal to the subspace.

Let W be a subspace of $\mathbb{R}^n$. $\forall y \in \mathbb{R}^n$, $!\exists(\hat{y}, z) \in W \times W^{\perp}$ such that:

$$y = \hat{y} + z$$

[15] Nora Belrose, first author of the paper, explained LEACE very clearly in this video: https://www.youtube.com/watch?v=y6Z8CYZz1eo

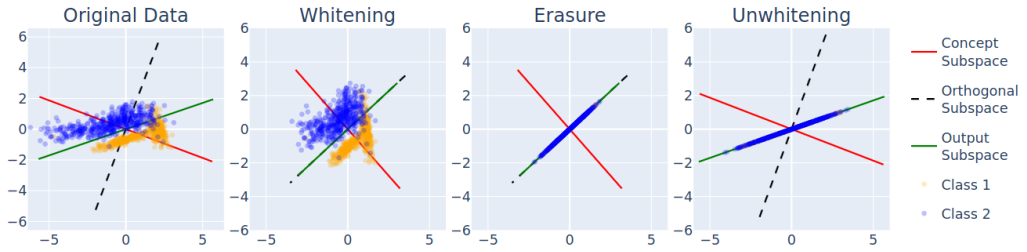

Figure 41: LEACE method illustrated. Blue and orange points represent activation vectors projected in a 2D space. The goal of LEACE is to make the network incapable of distinguishing between two classes of the concept. To achieve this, LEACE works in three steps. 1) It identifies a "concept subspace" shown as a red line, formed by connecting the centroids of each class. The dashed line represents the subspace that is orthogonal to the concept subspace. Projecting the data onto this orthogonal subspace erases the concept but not in the most surgical way possible. This method becomes the most precise form of erasure when the data has equal variance in all directions. This is what the **whitening** operation does: it transforms the data such that it has equal variance in all directions. 2) The **erasure** is performed through projection on the "orthogonal subspace". The green line ("output subspace") represents the erasure direction. 3) Following erasure, LEACE finishes with an **unwhitening** operation. [5].

Belrose et al. applied LEACE to remove gender information from a BERT model. To do this, they used a dataset of short biographies annotated with both gender and profession labels. Each biography was embedded, and the activations at the last layer, on the `[CLS]` token, were considered[16]. Each biography is associated with an activation vector, and a linear classifier was trained on these activation vectors to predict the individual's profession.

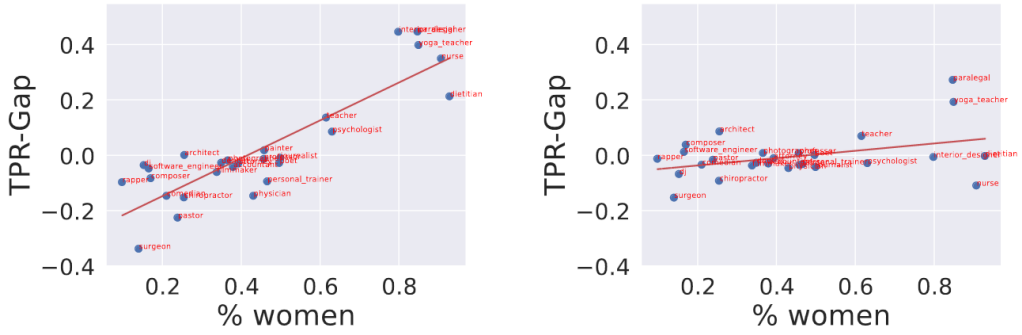

Figure 42: Removal of gender information from a BERT model. The TPR-Gap, which stands for True Positive Rate Gap, serves as a gender bias metric. Before erasure (left plot), the activations of the `[CLS]` token allowed for a relatively accurate prediction of an individual's profession ($R = 0.867$). After gender information erasure (right plot), it becomes more intricate ($R = 0.392$) [5].

LEACE is performed on one layer at a time. To guarantee that the model does not rely at all on the concept being erased, erasure can be done at every intermediate layer. This method is known as **concept scrubbing**. Erasures have to be performed sequentially from the first layer because applying LEACE at layer l impacts the distribution of intermediate representations in subsequent layers. Note that LEACE can also be applied to transformers' residual streams.

[16]The `[CLS]` token is unique to BERT, and was introduced because BERT is trained on two tasks. 1) BERT is trained to predict masked tokens in sentences (which is not necessarily the next one in the sentence, as in a transformer). 2) Half of the time during training BERT is given two sentences and has to predict whether the second one follows the first one. The `[CLS]` (for "classification") token was introduced to perform this task. The `[CLS]` token, located at the beginning of a sentence, serves to represent the sentence's overall meaning, which helps sentence-level classification.

11 Critics of Interpretability

Tools such as feature visualization and pixel attribution techniques like *Grad-CAM* seem aesthetically appealing, but their practical application, especially in industrial contexts, remains limited. Numerous authors have criticized interpretability [30]. This section invites a reconsideration of the role and efficiency of interpretability in machine learning. While it possesses certain merits, its current application and theoretical underpinnings warrant a more critical and nuanced examination. Currently, successful auditing of deception in ML models, such as CICERO's strategic deception, relies only on prompt engineering and monitoring [26], but does not rely on interpretability.

11.1 General Critics of Interpretability

A Long List of Theories of Impact for Interpretability [1], from Neel Nanda enumerates 20 diverse theories regarding the impact of interpretability. A critical examination of these theories reveals several fundamental problems:

1. **Preferential Alternatives to Interpretability**: often, objectives sought through interpretability are more efficiently achieved through other means. This is exemplified in the realm of deceptive alignment, where interpretability primary focus is currently outperformed by alternative strategies, as exemplified by [26].

2. **Too many goals**: Interpretability endeavors to simultaneously address multiple goals, which can dilute its effectiveness. It is generally more productive to target each sub-objective individually rather than amalgamating them into a single objective.

3. **Potential Adverse Effects**: While interpretability can contribute positively to safety, it also holds the potential to inadvertently bolster system capabilities, leading to unintended consequences.

Lesser disagreements include the overshadowing of conceptual advancements by interpretability and its predominant use in post-hoc analysis rather than proactive prediction or prevention.

The dissonance with key theories, particularly those centered around predicting future systems and auditing for deception, further accentuates the need for a reevaluation of the interpretability paradigm.

11.2 Could Interpretability be Used for Predicting Future Machine Learning Systems Abilities?

Neel Nanda suggests that interpretability can enhance our understanding of future machine learning systems. However, this theory has not been truly verified. For instance, the development of in-context learning in models [24], often cited as evidence of rapid capability gains during training, was primarily identified through behavioral evaluations and only then identified by interpretability techniques with the study of induction heads. This example underlines a critical point: interpretability often follows the discovery of phenomena, rather than predicting it. Moreover, interpretability has not really provided insights that allow for the extrapolation of future systems' behaviors or capabilities.

11.3 The Challenge of Auditing for Deception Using Interpretability

Interpretability could be a tool for detecting deception within models. The idea is that by examining random bits of a model to identify specific circuits or features, one might uncover deceptive elements. However, the practicality of this approach is questionable. For example, reverse-engineering a specific circuit in GPT-2 does not straightforwardly translate to an understanding of deceptive mechanisms within the model. Even if interpretability could focus on circuits relevant to deception or social modeling, it remains unclear whether such an analysis of parts of the model would be sufficiently comprehensive to counteract deception.

Alternative approaches to studying deception in AI are gaining traction. These include adversarial attack strategies, the creation of models specifically for studying deceptive alignment [17], and the identification of proxies for deceptive alignment. These methods, along with others like neural distillation and speed priors, may provide more tangible results in understanding and countering deception

in AI systems. The urgency of conceptual advances in understanding AI deception is emphasized over the current focus on interpretability. Theoretical frameworks like the *Simulator Theory* offer significant advancements in our understanding of deceptive alignment in AI while not using interpretability. A shift towards more proactive and theoretically grounded approaches may yield better insights into the future development and ethical alignment of AI systems.

11.4 Enumerative Safety may be Fundamentally Flawed

Enumerative safety, as conceptualized by Neel Nanda, proposes the ambitious goal of identifying and analyzing every feature within a model to detect those linked to dangerous capabilities or intentions. This approach, however, appears intrinsically flawed for several reasons.

Firstly, the task of determining the dangerousness of a feature in a model is inherently mis-specified. The quest to pinpoint dangerous features within the weights and structures of a neural network seems futile, as features in isolation do not possess inherent moral value. Their potential for harm or benefit largely depends on their contextual application within the system. For example, a feature in a neural network layer that visually resembles a weapon does not necessarily signify that the system poses a threat. Instead, it might simply indicate the system's ability to recognize objects that could be dangerous in certain contexts.

Moreover, the actual risks posed by models often emanate not from low-level features but from high-level behavioral abilities. These include advanced capabilities such as coding, exhibiting sycophantic behaviors, or possessing various forms of situational awareness and various theory of mind properties. Such complex abilities are usually composed of multiple interdependent sub-features, and their removal could detrimentally affect the model's overall functionality and performance.

The concept of deep deceptiveness further illustrates the complexity of identifying risks within AI systems. It posits that a system could exhibit deceptive behaviors not because of any single dangerous component, but due to the collective interaction of various elements within the model and its environment.

Previous attempts to apply the strategy of enumerative safety, particularly in the field of vision via automatic interpretability techniques, have shown limited success. Techniques such as *Network Dissection* [3] or *Compositional Explanations of Neurons* [18], and others have offered valuable insights but have not significantly impacted our approach to enhancing the robustness and safety of AI systems. Notably, many neurons in these systems elude straightforward interpretation, illustrating the complexity and limitations of current interpretability techniques.

Nonetheless, recent progress in enumerative safety [34] makes this path one of the most attractive ones to understand models. In summary, while the goal of enumerative safety in machine learning is ambitious and seemingly logical, its practical application is fraught with conceptual and technical challenges.

12 Further Readings

We were unable to include many important papers either because they were published during the writing of this manuscript or because, while significant, they were not as foundational as other papers we have presented. This chapter can only serve as an introduction. Here is a list of papers that we believe would merit inclusion in a second edition:

1. **Representation Engineering and Activation Vectors Literature**: These significant papers demonstrate how to modify the behaviors of GPTs by adding vectors into the residual stream without changing the neural network's weights. This allows for straightforward modifications to high-level behaviors. For example, it is possible to find a vector that makes the model more "Honest" or "Less polite". Papers and literature include: [38] and the Activation Vectors literature.

2. **Techniques for Concept Erasure Using Probing**: Linear probing can be an excellent technique to verify that the neural network encodes a concept linearly. RLACE is a superior technique that identifies a vector which, when projected orthogonally, causes the disappearance of information (i.e., the accuracy of a probing decreases maximally). [28]

3. **Automatic Circuit Discovery (ACDC) and Causal Scrubbing**: A major critique of many interpretability techniques is their non-causal nature. To resolve these issues and to aid in automatically interpreting neural networks and creating better methodologies, causal methods are being developed. References include: ACDC [10], Causal Scrubbing, and other causal methods (Causal Methodologies Summary).

4. **Macroscopic Interpretability**: The paper *Look Before You Leap: A Universal Emergent Decomposition of Retrieval Tasks in Language Models* offers insights into how LLMs decompose retrieval tasks modularly, particularly addressing how middle layers at the last token position process the request, while late layers retrieve the correct entity from the context [35].

5. **Developmental Interpretability**: It is an emerging research area in AI alignment that studies the formation of structures within neural networks. It draws analogies with developmental biology, focusing on understanding the neural network's evolution during training. The research particularly emphasizes identifying and interpreting phase transitions throughout training. Read Towards Developmental Interpretability and check the devinterp website for more.

6. **SolidGoldMagikarp**: This represents astonishing results showing that neural networks contain weird failure modes. For instance, some tokens like 'SolidGoldMagikarp' are present in the transformer's tokenizer but not in the corpus [29].

7. **Ex-Ante Training Methods in Interpretability**: The majority of presented papers in this chapter discuss post-training techniques. However, creating architectures interpretable by design could be highly beneficial for safety and could make interpretability easier. An example of more intrerpretable architecture is the paper *Seeing is Believing: Brain-Inspired Modular Training for Mechanistic Interpretability* [14] : "*we embed neurons in a geometric space and augments the loss function with a cost proportional to the length of each neuron connection. [...] We demonstrate that BIMT discovers useful modular neural networks for many simple tasks, revealing compositional structures in symbolic formulas*". This review paper [27] discusses several other methods for creating sparser architectures, which facilitate interpretability.

8. **World Models in Neural Networks**: Papers like OthelloGpt [19] shows that transformers trained for specific tasks like Othello contain linear representations of the game board, challenging previous notions of nonlinear representations. The paper [12] shows that Llama-2 contains a detailed model of the world by being able to find representation of longitude and latitude of different countries. Other examples are presented in the paper *Eight things to know about large language models* [6].

9. **Knowledge Representation**: Investigating how language models represent facts is crucial. While only the ROME paper was presented in this chapter, Fact Finding is another paper that attempts to reverse-engineer factual recall at the neuron level.

10. **Discovering Latent Knowledge**: *Discovering latent knowledge in language models without supervision* [9] tries to find "the direction of truth" in an unsupervised manner. Although the current techniques have limitations, future models with more unified internal representations might enable the use of this technique more effectively.

The reason why this chapter was specifically dedicated to transformers rather than other LLM architectures is because the majority of papers in interpretability which are relevant for AGI safety focus on them, but most of the results should transfer to other LLM architectures trained on an autoregressive way on a tokenized corpus.

References

[1] *A Longlist of Theories of Impact for Interpretability — LessWrong*. URL: https://www.lesswrong.com/posts/uK6sQCNMw8WKzJeCQ/a-longlist-of-theories-of-impact-for-interpretability (visited on 12/20/2023).

[2] *A Mathematical Framework for Transformer Circuits*. URL: https://transformer-circuits.pub/2021/framework/index.html (visited on 06/19/2023).

[3] David Bau et al. "Understanding the role of individual units in a deep neural network". In: *Proceedings of the National Academy of Sciences* 117.48 (Dec. 2020). Publisher: Proceedings of the National Academy of Sciences, pp. 30071–30078. DOI: 10.1073/pnas.1907375117. URL: https://www.pnas.org/doi/full/10.1073/pnas.1907375117 (visited on 12/20/2023).

[4] Nora Belrose et al. *Eliciting Latent Predictions from Transformers with the Tuned Lens*. arXiv:2303.08112 [cs]. Aug. 2023. DOI: 10.48550/arXiv.2303.08112. URL: http://arxiv.org/abs/2303.08112 (visited on 09/08/2023).

[5] Nora Belrose et al. *LEACE: Perfect linear concept erasure in closed form*. arXiv:2306.03819 [cs]. June 2023. DOI: 10.48550/arXiv.2306.03819. URL: http://arxiv.org/abs/2306.03819 (visited on 07/10/2023).

[6] Samuel R Bowman. "Eight things to know about large language models". In: *arXiv preprint arXiv:2304.00612* (2023).

[7] Samuel R. Bowman. *Eight Things to Know about Large Language Models*. arXiv:2304.00612 [cs]. Apr. 2023. DOI: 10.48550/arXiv.2304.00612. URL: http://arxiv.org/abs/2304.00612 (visited on 12/28/2023).

[8] Tom B. Brown et al. *Language Models are Few-Shot Learners*. arXiv:2005.14165 [cs]. July 2020. DOI: 10.48550/arXiv.2005.14165. URL: http://arxiv.org/abs/2005.14165 (visited on 08/30/2023).

[9] Collin Burns et al. "Discovering latent knowledge in language models without supervision". In: *arXiv preprint arXiv:2212.03827* (2022).

[10] Arthur Conmy et al. "Towards automated circuit discovery for mechanistic interpretability". In: *arXiv preprint arXiv:2304.14997* (2023).

[11] Nelson Elhage. "Softmax Linear Units". In: *Transformer Circuits Thread* (2022). URL: https://transformer-circuits.pub/2022/solu/index.html.

[12] Wes Gurnee and Max Tegmark. "Language models represent space and time". In: *arXiv preprint arXiv:2310.02207* (2023).

[13] jacquesthibs. "But is it really in Rome? An investigation of the ROME model editing technique". en. In: (). URL: https://www.lesswrong.com/posts/QL7J9wmS6W2fWpofd/but-is-it-really-in-rome-an-investigation-of-the-rome-model (visited on 09/07/2023).

[14] Ziming Liu, Eric Gan, and Max Tegmark. "Seeing is Believing: Brain-Inspired Modular Training for Mechanistic Interpretability". In: *arXiv preprint arXiv:2305.08746* (2023).

[15] Callum McDougall. "Induction heads - illustrated". en. In: (). URL: https://www.lesswrong.com/posts/TvrfY4c9eaGLeyDkE/induction-heads-illustrated (visited on 12/20/2023).

[16] Kevin Meng et al. *Locating and Editing Factual Associations in GPT*. arXiv:2202.05262 [cs]. Jan. 2023. DOI: 10.48550/arXiv.2202.05262. URL: http://arxiv.org/abs/2202.05262 (visited on 09/02/2023).

[17] *Model Organisms of Misalignment: The Case for a New Pillar of Alignment Research — AI Alignment Forum*. URL: https://www.alignmentforum.org/posts/ChDH335ckdvpxXaXX/model-organisms-of-misalignment-the-case-for-a-new-pillar-of-1 (visited on 12/20/2023).

[18] Jesse Mu and Jacob Andreas. *Compositional Explanations of Neurons*. en. June 2020. URL: https://arxiv.org/abs/2006.14032v2 (visited on 12/20/2023).

[19] Neel Nanda, Andrew Lee, and Martin Wattenberg. "Emergent linear representations in world models of self-supervised sequence models". In: *arXiv preprint arXiv:2309.00941* (2023).

[20] Neel Nanda et al. *Progress measures for grokking via mechanistic interpretability*. en. arXiv:2301.05217 [cs]. Jan. 2023. URL: http://arxiv.org/abs/2301.05217 (visited on 07/11/2023).

[21] nostalgebraist. "interpreting GPT: the logit lens". en. In: (). URL: https://www.lesswrong.com/posts/AcKRB8wDpdaN6v6ru/interpreting-gpt-the-logit-lens (visited on 09/08/2023).

[22] Chris Olah et al. "The Building Blocks of Interpretability". In: *Distill* 3.3 (Mar. 2018), 10.23915/distill.00010. ISSN: 2476-0757. DOI: 10.23915/distill.00010. URL: https://distill.pub/2018/building-blocks (visited on 09/01/2023).

[23] Chris Olah et al. "Zoom In: An Introduction to Circuits". In: *Distill* 5.3 (Mar. 2020), 10.23915/distill.00024.001. ISSN: 2476-0757. DOI: 10.23915/distill.00024.001. URL: https://distill.pub/2020/circuits/zoom-in (visited on 06/19/2023).

[24] Catherine Olsson et al. *In-context Learning and Induction Heads*. arXiv:2209.11895 [cs]. Sept. 2022. DOI: 10.48550/arXiv.2209.11895. URL: http://arxiv.org/abs/2209.11895 (visited on 06/19/2023).

[25] Alexander Pan, Kush Bhatia, and Jacob Steinhardt. *The Effects of Reward Misspecification: Mapping and Mitigating Misaligned Models*. arXiv:2201.03544 [cs, stat]. Feb. 2022. DOI: 10.48550/arXiv.2201.03544. URL: http://arxiv.org/abs/2201.03544 (visited on 08/30/2023).

[26] Peter S. Park et al. *AI Deception: A Survey of Examples, Risks, and Potential Solutions*. arXiv:2308.14752 [cs]. Aug. 2023. DOI: 10.48550/arXiv.2308.14752. URL: http://arxiv.org/abs/2308.14752 (visited on 12/20/2023).

[27] Tilman Räuker et al. "Toward transparent ai: A survey on interpreting the inner structures of deep neural networks". In: *2023 IEEE Conference on Secure and Trustworthy Machine Learning (SaTML)*. IEEE. 2023, pp. 464–483.

[28] Shauli Ravfogel et al. "Linear adversarial concept erasure". In: *International Conference on Machine Learning*. PMLR. 2022, pp. 18400–18421.

[29] Jessica Rumbelow and Matthew Watkins. "SolidGoldMagikarp (plus, prompt generation)". In: *AI ALIGNMENT FORUM*. 2023.

[30] Charbel-Raphaël Segerie. "Against Almost Every Theory of Impact of Interpretability". en. In: (). URL: https://www.lesswrong.com/posts/LNA8mubrByG7SFacm/against-almost-every-theory-of-impact-of-interpretability-1 (visited on 12/20/2023).

[31] *The Illustrated Transformer – Jay Alammar – Visualizing machine learning one concept at a time.* URL: https://jalammar.github.io/illustrated-transformer/ (visited on 06/19/2023).

[32] *The Reversal Curse: LLMs trained on" A is B" fail to learn" B is A"*. URL: https://scholar.google.com/citations?view_op=view_citation&hl=en&user=4VpTwzIAAAAJ&sortby=pubdate&citation_for_view=4VpTwzIAAAAJ:K3LRdlH-MEoC (visited on 10/20/2023).

[33] *Thread: Circuits*. URL: https://distill.pub/2020/circuits/ (visited on 09/11/2023).

[34] *Towards Monosemanticity: Decomposing Language Models With Dictionary...* en. URL: https://www.anthropic.com/index/towards-monosemanticity-decomposing-language-models-with-dictionary-learning (visited on 12/08/2023).

[35] Alexandre Variengien and Eric Winsor. "Look Before You Leap: A Universal Emergent Decomposition of Retrieval Tasks in Language Models". In: *arXiv preprint arXiv:2312.10091* (2023).

[36] Ashish Vaswani et al. *Attention Is All You Need*. arXiv:1706.03762 [cs]. Dec. 2017. DOI: 10.48550/arXiv.1706.03762. URL: http://arxiv.org/abs/1706.03762 (visited on 06/19/2023).

[37] Kevin Ro Wang et al. "Interpretability in the Wild: a Circuit for Indirect Object Identification in GPT-2 Small". en. In: Sept. 2022. URL: https://openreview.net/forum?id=NpsVSN6o4ul (visited on 09/08/2023).

[38] Andy Zou et al. "Representation engineering: A top-down approach to ai transparency". In: *arXiv preprint arXiv:2310.01405* (2023).