University of Bergen
Department of Informatics

Analyzing Indirect Object Identification Circuits in TinyStories-8M

Author: Marius Binner
Supervisors: Pekka Parviainen

UNIVERSITETET I BERGEN
Det matematisk-naturvitenskapelige fakultet

May 13, 2025

Abstract

This thesis investigates the mechanisms underlying transformer language model behavior, focusing on the Indirect Object Identification (IOI) task in the TinyStories-8M model, comparing it against GPT2-small. Using established mechanistic interpretability techniques, we analyze how these models solve the IOI task and compare their internal circuits. Our findings reveal that TinyStories-8M, despite being a much smaller model trained on simpler data, implements an IOI circuit remarkably similar to that of GPT2-small. However, TinyStories-8M's circuit is notably cleaner and simpler, lacking for example the negative name mover and induction heads found in GPT2-small. We demonstrate that TinyStories-8M achieves in-context learning without relying on induction heads, which have been previously argued to be crucial for this capability. Through the application of sparse autoencoders, we identify specific features responsible for signaling name-moving behavior to attention heads within the IOI circuit. This provides deeper insights into the functioning of the IOI circuit and attention mechanisms in general, supporting the idea that features written into token positions can trigger specific attention behaviors. Our research contributes to the field of mechanistic interpretability by offering a comparative analysis of circuit formation in models of different scales and training regimes. It challenges existing assumptions about the necessity of certain architectural features for in-context learning and provides a foundation for future work in understanding the use of features as computational units in transformer circuits.

Contents

Introduction

Artificial intelligence, particularly in the form of large language models, has made remarkable strides in recent years. These models can now perform a wide range of tasks, from answering questions to generating coherent text, often with human-like proficiency. However, despite their impressive capabilities, the inner workings of these models remain largely opaque. This lack of interpretability poses significant challenges, both for improving the models and for ensuring their safe and ethical deployment.

Mechanistic interpretability is an emerging field that aims to address this challenge by reverse-engineering the specific computations and representations that neural networks use to perform their tasks. By developing an interpretable mechanistic understanding of how information flows and is processed within these networks, researchers hope to gain insights that can lead to more robust, efficient, and trustworthy AI systems.

This thesis focuses on applying mechanistic interpretability techniques to analyze and compare two language models: GPT2-small and TinyStories-8M. While both are transformer-based models capable of performing various language tasks, they differ significantly in size, training data, and intended purpose. GPT2-small is a general-purpose language model trained on a diverse corpus of web text, while TinyStories-8M is a smaller model trained on a dataset of simpel, synthetic stories designed for young children.

The primary objectives of this research are:

1. To investigate how these models solve the Indirect Object Identification (IOI) task, a benchmark problem in mechanistic interpretability that tests a model's ability identify the indirect object in a verb sentence.

2. To compare the internal mechanisms and circuits used by GPT2-small and TinyStories-8M in solving the IOI task, with a particular focus on identifying similarities and differences in their approaches.

3. To explore the use of gated sparse autoencoders as a tool for uncovering and interpreting the features and representations learned by these models.

4. To contribute to the broader understanding of how transformer-based language models process and manipulate information, particularly in the context of in-context learning.

By conducting this comparative analysis, we aim to shed light on how different training regimes and model architectures can lead to similar or divergent internal mechanisms for solving language tasks.

The thesis is structured as follows: Chapter 2 provides background information on mechanistic interpretability, transformer architectures, and the IOI task. Chapter 3 details the experimental methodology used in this study and the observations made. Chapter 4 summarizes the main results of our experiments. Finally, Chapter 5 discusses the implications of our findings and suggests directions for future research in this rapidly evolving field.

Background

2.1 Mechanistic Interpretability

Mechanistic interpretability is a field of study that aims to understand the inner workings of neural networks at an algorithmic level. It goes beyond traditional interpretability methods by seeking to reverse engineer the specific computations and representations that a model uses to perform its tasks. The goal is to develop an interpretable mechanistic understanding of how information flows and is processed within the network to perform a specific task. An example might involve taking the networks computational graph, ran on a certain input, explaining every step from input to output as one would explain a mathematically defined algorithm.

As an analogy, imagine trying figure out which interpretable mental steps a human goes through while multiplying two numbers by inspecting their brain's neural circuitry. Some advantages a mechinterp researcher has over a neuroscientist is that neural networks are perfectly deterministic, we can do perfect counterfactual experiments, and we have perfect read/write access to every step of the computation.

Examples of previous results include: explaining the grokking phenomenon for small transformers performing modular addition by explaining how they perform the task by using trig identities and the discrete Fourier transform to reduce the problem to rotations around a circle [Nanda et al., 2023], identifying and explaining circuits in GPT2-Small that allow it to perform indirect object identification [Wang et al., 2022], or analyses of larger models, like insight into how the 70 billion parameter Chinchilla model performs multiple choice question answering [Lieberum et al., 2023].

In the remainder of this section I will give an overview of foundational interpretability concepts that are needed for understanding the field.

2.1.1 Features

A central theme in neural network interpretability concerns understanding what features the network is representing, how these features are represented, and how how they're used to perform a task. But what are features? There's not really any established definition, but we can point to some central properties that will generally pick out what's meant by it in the context of mechanistic interpretability:

- Examples of features are: "is a giraffe", "is a digit", "is an even digit", "is Barrack Obama", "vertical edge", "is red", "man in a car", "protagonist of story", "pixel with color value RGB(88, 32,101)", "weird looking uninterpretable clusters of pixels".
- Features are the basic building blocks of the network's representations. They are the units of information that the network uses to make predictions.
- Features are abstractions of the input data. For example a dog detector network might take raw pixels as input and have a feature like "is a dog" or "floppy ear" that summarizes some property of the input.

- Features can activate in response to other features. For example the feature "is a dog" might activate in response to features like "has four legs" and "barks".
- Features can be more or less abstract. For example "a basket of 17 puppies" is more abstract than "vertical edge" or "patch of brown".
- Features can occur at different levels of intensity. For example some inputs can activate the "bird" feature more than others. There are more or less central category members and more central category members would tend to activate the "is a bird" feature more. For example we can imagine a dove activating the "bird" feature more than a penguin or a guy in a bird costume.

The previous points are not intended to have empty intersection nor be exhaustive, and are only meant to give an intuition for some of the various uses of the concept.

2.1.2 Linear representation hypothesis

How are features represented in neural networks? A priori they could be represented in many different ways: perhaps the presence of a "dog" feature in a layer is represented by the activation vector existing on a certain non-linear manifold in activation space, or perhaps there are convex polytopes in activation space that correspond to features like is argued in [Black et al., 2022], or maybe it's a completely incomprehensible discontinuous set of points, or maybe features aren't represented in single layers and they only exist jointly in the network as a whole. However, a common hypothesis in mechinterp is that features are represented linearly in the activation space in each of the network's layers [Elhage et al., 2022]. This means that the activation vector of a layer can be thought of as a linear combination of the features it represents. I will assume this hypothesis is mostly true.

Assumption (Linear representation hypothesis). *The activation vector of a layer can be represented as a linear combination of the features it represents:*

$$\mathbf{a} = \sum_i \alpha_i \mathbf{f}_i$$

where $\mathbf{a}$ *is the activation vector,* $\mathbf{f}_i$ *are the unit feature directions, and* α_i *are strength of the feature activations. This means that features are directions in activation space.*

This is not to say that the activations are linear functions of the input (they're obviously not), only that the activation vector in a layer is a linear function of features represented in that layer. How interesting this hypothesis is depends on how you interpret it and what you intend to do with it: since the features themselves are non-linear functions of the input, given the right choice of features any activation vector could be represented as a linear combination of features. The hypothesis is interesting if the features are in some sense interpretable. For example if we could decompose an activation vector into a linear combination of "has four legs", "has floppy ear", "is brown", etc. This is an important point to keep in mind.

There are some arguments for the linear representation hypothesis in [Elhage et al., 2022], for example the fact that the vast majority of computation in neural networks are matrix multiplications could indicate that it's a natural format for the network to use. Importantly, there's also a large amount of empirical support for it in the case of transformer language models [Templeton, 2024], [Bricken et al., 2023] [Cunningham et al., 2023]. However, there have been some counter examples to it showing that it's not true in all cases, for example in [Engels et al., 2024] where circular features are used to represent certain units of time. It can be tricky to make a solid case for non-linear features though, since as mentioned earlier, it depends on choice of features and you could always imagine that there are some clean linear representations you've not found.

Given the previous considerations it's important to be careful when using the linear representation hypothesis. It's not a strict mathematical law, but rather a largely true phenomenon, at least for real world transformer language models. If it's the case that most of the features are represented linearly this is sufficient for understanding a lot of the network's behavior. So going forward I will assume that features are represented as directions in activation space.

2.1.3 Polysemanticity and superposition

A neural network neuron is said to be **monosemantic** if it represents a single feature. This means that, if you take the set of all features represented in the corresponding layer, all but one of them are orthogonal to the coordinate axis corresponding to that neuron. If all the neurons in a layer are monosemantic we call the layer monosemantic. This implies a monosemantic layer with n neurons can represent at most n features. If a neuron is not monosemantic it's called **polysemantic**, which means that it can activate in response to multiple features. The presence of polysemantic neurons could be for one of two reasons:

1. There is no nothing special about the neuron directions that would make it so the network would want to represent features along those directions. This is called a **non-privileged basis**. In the opposite case we'd say the network has a **privileged basis**.

2. There are more features than there are neurons in a layer, forcing the network to have features ”share” neurons. This is called **superposition**.

We can also obviously have 1. and 2. occur at the same time.

An example of a privileged basis is the internal activation vector of an MLP. This is because a non-linear function is applied coordinate-wise, making the coordinate axes special. An example of a non-privileged basis is the one that represents the word embedding vectors. They don't have a bias towards being represented in any given basis since the weight matrices reading them could could just learn to rotate them into any other basis. In a non-privileged basis there's no reason to expect the neurons to be monosemantic. The previous points about privileged bases refer to architectural properties of the network. We could still expect a privileged basis for other reasons, for example due to the optimization process, initialization, data distribution, or loss function. For instance a L1 loss placed on the activations. It's also observed that the Adam optimizer tends to incentivize a privileged basis due to the dimension-wise normalization [Elhage et al., 2023].

Empirically, neurons in many neural networks have been observed to be highly polysemantic [Olah et al., 2020], [Elhage et al., 2022], which is considered a major challenge for interpretability work. If neurons activate in response to multiple features, it becomes difficult to understand what a neuron is doing at any given time. What we fundamentally care about is which features are activating, since this is what allows us to interpret what meaningful information the network is representing. If we knew the feature directions and no superposition was taking place, we could recover the feature activations by just solving a linear system. But in the case of superposition the feature directions give us an overcomplete basis for the activation space and thus an underdetermined system, so any activation vector can be represented as infinitely many linear combinations of features. So then, which is the correct one? Much work has been put into answering this question [Bricken et al., 2023], [Templeton, 2024]

[Elhage et al., 2022] gives a detailed study of superposition in a toy model, detailing much insight into why, how and when superposition occurs. Some of the key insights are:

- There are far more features that are useful for the network to represent than it has neurons.

- If all the features are orthogonal you can only represent n of them. However, if you allow a small bounded inner product between features directions, the number you can represent is an exponential function of n (Johnson–Lindenstrauss lemma).
- Due to the fact that most features are sparse, i.e. low probability of activating at any given time, sparse dictionary learning can be used to find feature directions.
- Features that more sparse are more likely to share dimensions with other features, since they're less likely to interfere. Features that are less sparse are more likely get dedicated dimensions.
- Features that are less relevant for decreasing the loss function are more likely share dimensions, and those that are more relevant are more likely to get a dedicated dimension.
- Features that correlate are more likely to be orthogonal since the chance of interference is greater, or when that's difficult they might be merged into a single feature similar to what happens in principal component analysis.
- Anti-correlated features more likely to be represented in anti-podal pairs, since they have far lower chance of interference with each other.

One hypothesis of the relevance of sparsity is that the network wants to reduce interference: if 2 features have high inner product and activate at the same time they will interfere with each other, amplifying each others signal, making it difficult for the network to know to which extent the signal is supposed to be there. While interference with non-present features will still produce slight activation in them, it's easier for the network to ignore it as "noise". Or in a worse case, when the inner product is negative, they'll destructively interfere with each other's signal.

A central idea is that networks with superposition are trying to emulate the behavior of a hypothetical larger network that has enough neurons to dedicate one for each feature the network wants to represent. It's this type of idea that motivates many techniques for undoing superposition, like sparse autoencoders [Bricken et al., 2023].

2.2 GPT style transformers

The transformer architecture was famously introduced in [Vaswani et al., 2017], where it consisted of both encoder and decoder blocks. In [Radford et al., 2018], GPT style transformers were introduced that uses only the decoder part. It's the latter kind of architecture I'll have a look at. The remainder of this section will give a mathematical description of the transformer implementation I'll be interpreting.

2.2.1 All the pieces

Autoregressive Language Modeling

The problem setup is as follows: we have a corpus of text sampled from some distribution that we tokenize into a sequence $x_1, x_2, \ldots, x_T$ where x_t is a token $t_i \in D$ and D (called the dictionary) is a finite indexed set of the possible tokens that could exist in a sequence. The goal is to learn a probability distribution $p(X_t|X_{t-1}, X_{t-2}, \ldots, X_{t-k})$ that accurately predicts the next token in a sequence given the k previous tokens. In my case p will be a transformer model and k is its context size.

Input & Output

The raw output of the transformer will be a vector of logits $\mathbf{z} \in \mathbb{R}^{|D|}$ that we can convert to a probability distribution using the softmax function:

$$p(X_t = t_i | x_{t-1}, x_{t-2}, \ldots, x_{t-k}) = \frac{\exp(z_i)}{\sum_{j=1}^{|D|} \exp(z_j)}$$

where i is the position in the vector $\mathbf{z}$ corresponding to the ith token t_i in D. The input to the transformer will be a matrix of one-hot encoded vectors such that each row in the matrix corresponds to a token in the sequence in the corresponding order:

$$\mathbf{X} = \begin{bmatrix} \mathbf{e}_1 \\ \mathbf{e}_2 \\ \vdots \\ \mathbf{e}_T \end{bmatrix} \in \mathbb{R}^{T \times |D|}$$

where $\mathbf{e}_t = [0, 0, \ldots, 1, \ldots, 0]$ is a one-hot encoded row vector of the token x_t, consisting of all zeros except for a one at the position corresponding to the token x_t, i.e. at position i such that $t_i = x_t$.

Tokenization

Tokenization is the process of converting raw text into a sequence of tokens that the model can process. This step is crucial as it determines how the model "sees" the input text. The choice of tokenization strategy can significantly impact the model's performance and interpretability. Common tokenization methods include:

1. Word-level tokenization: Each word is a separate token. This can lead to large vocabularies and issues with out-of-vocabulary words.
2. Subword tokenization: Words are broken into subword units. This balances vocabulary size and ability to handle rare words. Examples include Byte-Pair Encoding (BPE) and WordPiece.
3. Character-level tokenization: Each character is a separate token. This results in a small vocabulary but longer sequences.

GPT models typically use subword tokenization. This choice impacts how the model learns to represent and process language. For example, with subword tokenization:

- Common words are usually single tokens.
- Rare words are split into multiple subword tokens.
- The model can potentially learn morphological patterns.

Understanding the tokenization is crucial for interpreting the model's behavior, as it affects how semantic and syntactic information is encoded and processed throughout the network.

Embedding

The embedding layer transforms the one-hot encoded input into a dense vector representation. This is done by multiplying the input with an embedding matrix $\mathbf{W}_E \in \mathbb{R}^{|D| \times d_{\text{model}}}$, where d_{model} is the dimensionality of the embedding:

$$\mathbf{E} = \mathbf{X}\mathbf{W}_E$$

Each row of $\mathbf{W}_E$ corresponds to the embedding vector for a specific token in the dictionary. The embedding layer learns these representations during training.

Positional Encoding

In GPT-style transformers, learned positional encodings are often used instead of the fixed sinusoidal encodings from the original transformer paper. The learned positional encoding is a matrix $\mathbf{P} \in \mathbb{R}^{T \times d_{\text{model}}}$. Each row of $\mathbf{P}$ corresponds to a position in the sequence and is learned during training. The positional encoding is added to the token embeddings:

$$\mathbf{E}' = \mathbf{E} + \mathbf{P}$$

Learned positional encodings allow the model to capture more flexible position-dependent patterns, potentially adapting to specific characteristics of the training data or task.

Layer Normalization

Layer Normalization is applied to the input of each sub-layer in the transformer. It normalizes the inputs across the features, applying a transformation of the form:

$$\text{LayerNorm}(\mathbf{x}) = \gamma * \frac{\mathbf{x} - \mu}{\sqrt{\sigma^2 + \epsilon}} + \beta$$

where:

- $\mathbf{x}$ is the input vector
- μ and σ are the mean and standard deviation of the input computed across the feature dimension
- γ and β are learnable parameters
- ϵ is a small constant for numerical stability

Multi-Head Self-Attention

Multi-Head Self-Attention allows the model to move information between token positions.

$$\begin{aligned} \mathbf{X}_{\text{result}} &= \sum_{i=1}^{n_{\text{head}}} \mathbf{H}_{i,:,:} \mathbf{W}_O^i + \mathbf{b}_O \\ \text{where } \mathbf{H}_{i,:,:} &= \text{Attention}(\mathbf{Q}\mathbf{W}_Q^i + \mathbf{b}_Q^i, \mathbf{K}\mathbf{W}_K^i + \mathbf{b}_K^i, \mathbf{V}\mathbf{W}_V^i + \mathbf{b}_V^i) \\ \text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) &= \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_{\text{head}}}}\right)\mathbf{V} \end{aligned}$$

Here:

- $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{T \times d_{\text{model}}}$ are the query, key, and value matrices
- $\mathbf{W}_Q^i, \mathbf{W}_K^i, \mathbf{W}_V^i \in \mathbb{R}^{d_{\text{model}} \times d_{\text{head}}}$ are learned linear transformations for each head
- $\mathbf{b}_Q^i, \mathbf{b}_K^i, \mathbf{b}_V^i \in \mathbb{R}^{d_{\text{head}}}$ are learned bias vectors for each head

- $\mathbf{W}_O^i \in \mathbb{R}^{d_{\text{head}} \times d_{\text{model}}}$ are learned linear transformations for combining the outputs of each head
- $\mathbf{b}_O \in \mathbb{R}^{d_{\text{model}}}$ is a learned bias vector for the output
- $\mathbf{H}_{i,:,:} \in \mathbb{R}^{T \times d_{\text{head}}}$ represents the output of the i-th attention head
- $\mathbf{X}_{\text{result}} \in \mathbb{R}^{T \times d_{\text{model}}}$ is the final output of the multi-head attention layer
- n_{head} is the number of attention heads
- $d_{\text{head}} = d_{\text{model}}/n_{\text{head}}$ is the dimensionality of each head
- T is the sequence length

In GPT-style models, the attention mechanism is causal, meaning that each token can only attend to itself and the tokens that came before it. This is typically implemented by masking the attention scores before the softmax operation:

$$\text{CausalAttention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_{\text{head}}}} + \mathbf{M}\right)\mathbf{V}$$

where $\mathbf{M}$ is an upper triangular mask of $-\infty$ values:

$$\mathbf{M}_{ij} = \begin{cases} 0 & \text{if } i \geq j \\ -\infty & \text{otherwise} \end{cases}$$

This ensures that the softmax operation assigns zero probability to attending to future tokens.

MLP Layers

The MLP (Multi-Layer Perceptron) layers in a transformer consist of two linear transformations with a non-linear activation function in between. They are applied to each position separately and identically. The computation can be expressed as:

$$\begin{aligned} \mathbf{X}' &= \text{GELU}(\mathbf{X}\mathbf{W}_1 + \mathbf{b}_1) \\ \mathbf{X}_{\text{mlp}} &= \mathbf{X}'\mathbf{W}_2 + \mathbf{b}_2 \end{aligned}$$

where:

- $\mathbf{X} \in \mathbb{R}^{T \times d_{\text{model}}}$ is the input to the MLP layer
- $\mathbf{W}_1 \in \mathbb{R}^{d_{\text{model}} \times d_{\text{mlp}}}$ and $\mathbf{W}_2 \in \mathbb{R}^{d_{\text{mlp}} \times d_{\text{model}}}$ are learned weight matrices
- $\mathbf{b}_1 \in \mathbb{R}^{d_{\text{mlp}}}$ and $\mathbf{b}_2 \in \mathbb{R}^{d_{\text{model}}}$ are learned bias vectors
- d_{mlp} is the dimensionality of the feed-forward layer (typically larger than d_{model})
- GELU (Gaussian Error Linear Unit) is the activation function, defined as:

$$\text{GELU}(x) = x \cdot \Phi(x)$$

  where $\Phi(x)$ is the cumulative distribution function of the standard normal distribution

Decoder blocks

The decoder block is a key component of the transformer architecture. It consists of two main sub-layers: one multi-head self-attention layer and one MLP layer. Each sub-layer is preceded by a layer norm. The decoder block is applied multiple times in the model to process the input sequence. The computational graph of a decoder block is shown in Figure 2.1.

Unembedding

The unembedding layer is the final transformation that converts the model's internal representations back into logits over the vocabulary. It's essentially the inverse operation of the embedding layer. The unembedding is computed as:

$$\mathbf{z} = \mathbf{x}\mathbf{W}_U + \mathbf{b}_U$$

where:

- $\mathbf{x} \in \mathbb{R}^{d_{\text{model}}}$ is the final representation of a token
- $\mathbf{W}_U \in \mathbb{R}^{d_{\text{model}} \times |D|}$ is the unembedding matrix
- $\mathbf{b}_U \in \mathbb{R}^{|D|}$ is a learned bias vector
- $\mathbf{z} \in \mathbb{R}^{|D|}$ is the output logits vector

In many implementations, the unembedding matrix $\mathbf{W}_U$ is tied to the transpose of the embedding matrix $\mathbf{W}_E$ to reduce the number of parameters and potentially improve generalization:

$$\mathbf{W}_U = \mathbf{W}_E^T$$

The resulting logits $\mathbf{z}$ are then passed through a softmax function to obtain the final probability distribution over the vocabulary, as described in the Input & Output section.

2.2.2 Putting the pieces together

Now that we have all the pieces, we can put them together to define the full model. The forward pass of a GPT-style transformer for autoregressive language modeling can be summarized as follows:

Input Embedding:

$$\mathbf{E} = \mathbf{X}\mathbf{W}_E$$

Add Positional Encoding:

$$\mathbf{E}' = \mathbf{E} + \mathbf{P}$$

Decoder block (repeated N times, starting with $\mathbf{X}_{\text{pre}} = \mathbf{E}'$):

$$\begin{aligned}
\mathbf{X}_{\text{mid}} &= \mathbf{X}_{\text{pre}} + \text{MultiHeadAttention}(\text{LayerNorm}(\mathbf{X}_{\text{pre}}) \\
\mathbf{X}_{\text{post}} &= \mathbf{X}_{\text{mid}} + \text{MLP}(\text{LayerNorm}(\mathbf{X}_{\text{mid}}))
\end{aligned}$$

Final Layer Normalization:

$$\mathbf{X}_{\text{final}} = \text{LayerNorm}(\mathbf{X}_{\text{post}})$$

Unembedding:

$$\mathbf{z} = \mathbf{X}_{\text{final}}\mathbf{W}_U + \mathbf{b}_U$$

Softmax:

$$p(X_t = t_i | x_{t-1}, x_{t-2}, \ldots, x_{t-n_{\text{context}}}) = \frac{\exp(z_i)}{\sum_{j=1}^{|D|}\exp(z_j)}$$

This process is applied autoregressively, meaning that for each position in the sequence, the model only has access to the previous tokens. The causal attention mask in the MultiHeadAttention layer ensures this property. The model is trained to minimize the negative log-likelihood of the target tokens given the previous tokens in the sequence. During inference, the model can generate new tokens by sampling from the output probability distribution and feeding the sampled token back as input for the next step.

2.3 A framework for thinking about transformers

[Elhage et al., 2021] gives a powerful conceptual framework for thinking about what the decoder only transformer is doing when it's performing tasks, which has been the foundation for much of the work in transformer mechinterp. In this section I give an overview of the main ideas.

2.3.1 Simplifications

For the purposes of this section I will make some simplifications to the transformer model. I will ignore the bias terms and the layer normalization. This is to make the math simpler and focus on some core operations that are happening in the transformer.

2.3.2 The residual stream

Since every layer (attention or MLP) of the transformer is paired with a residual connection, we can imagine that the input data flows directly to the output and is only modified along the way by having some vectors added to it:

$$\mathbf{X}' = \mathbf{X} + f(\mathbf{X})$$

We can imagine this as being a communication channel that's being read from and written to at each layer. Following [Elhage et al., 2021], we call this the *residual stream.* The residual stream at each step is a matrix $\mathbf{X} \in \mathbb{R}^{T \times d_{\text{model}}}$ containing one row vector for each token position. d_{model} is sometimes referred to as the residual stream dimension.

At the end of the residual stream the residual stream vector at each token position is multiplied by the unembedding matrix

$$\mathbf{X}\mathbf{W}_U$$

producing logit vectors corresponding to the next-token prediction at that position. This final multiplication is essentially a series of inner products between the residual stream vectors and the columns of the unembedding matrix. This creates an interpretation where different directions in the vector space of the residual stream correspond to different tokens in the vocabulary. For the network to produce a correct next-token prediction, some layers must write into the residual stream to increase magnitude along the right direction while keeping the other directions small enough.

Due to the residual connections we can decompose the final residual stream vectors into a sum of the initial embeddings and all the vectors added into the stream between the input and output. This can be helpful in trying to understand where the relevant magnitudes in the residual stream come from.

Non-privileged basis

Since every read/write operation is preceded by a linear transformation we can conclude that the network architecture doesn't privilege any basis for representing information in the residual stream. This means that the network could just learn a rotation that sends any basis into any other basis. Although the network architecture itself doesn't impose a bias for any specific bases, other factors like initialization or the optimization scheme might do this. However, the MLP layers do have privileged bases due to the component-wise activation functions.

Information movment & processing

Except for the attention layers, there is no communication between the token positions and all processing is done position-wise. Mathematically this is seen by noticing that the residual stream is only acted on by right multiplication

$$\mathbf{XW}$$

thus transforming the residual stream vector at each position independently, while the attention pattern is multiplied from the left

$$\mathbf{AX}$$

thus mixing information across positions. This assumes that token positions correspond to rows. Other than matrix multiplication, element-wise scaling/addition, and softmax (which is part of the attention pattern), the only operation done is layer normalization, which is done position-wise, normalizing along the residual stream dimension.

2.3.3 Attention heads

There are some observations worth making regarding the ways attention heads operate:

- Due to the causal masking, attention heads can only move information from left to right in the token positions.
- The weight matrices computing the attention patterns for each head are usually low rank compared with the residual stream dimension, typically $d_{\text{model}} = n_{\text{head}} * d_{\text{head}}$. This means that attention heads can only read/write from/to lower dimensional subspaces of the residual stream, allowing them to limit the amount of interaction they have with each other.
- The softmax accentuates the differences between the attention scores, making the attention patterns more sparse. This means that the heads will be biased towards gathering more of its information from fewer token positions at a time.
- Attention heads are independent and additive, meaning that the result of each attention head is added independently into the residual stream, allowing us, to some extent, interpret each head in the same layer separately. Although it could certainly be the case that the training process set up the heads to coordinate in certain ways. For example if a writing operation requires more bandwidth than the low rank of the head allows, the heads might have to coordinate to achieve this.

- Although the primary action of attention heads is information movement, they can also perform some processing via the $\mathbf{W}_V$ and $\mathbf{W}_O$ matrices, however limited due to the low rank. It's these matrices that decide which subspaces in the residual stream to read from/to, while it's the $\mathbf{W}_Q$ and $\mathbf{W}_K$ matrices that decide which token positions to move information from/to.
- Attention heads have a highly linear structure. Except for the softmax, they can be understood as entirely linear objects. If we hold the attention pattern constant, then they are entirely linear.

These observations point to a methodology of understanding transformers by imagining that each attention head is a separate entity implementing some simple information moving behavior that we might be able to interpret.

The OV and QK matrices

The matrix written into the residual stream by an attention head is

$$\mathbf{A}\mathbf{X}\mathbf{W}_V\mathbf{W}_O$$

where the lower triangular matrix $\mathbf{A} \in \mathbb{R}^{T\times T}$ is the attention pattern and $\mathbf{X}$ is the residual stream that the head read from. Since $\mathbf{W}_V$ and $\mathbf{W}_O$ are just matrices being multiplied we can simplify the analysis of the write operation by combining them into one matrix: $\mathbf{W}_V\mathbf{W}_O = \mathbf{W}_{OV} \in \mathbb{R}^{d_{\text{model}}\times d_{\text{model}}}$ which has rank d_{head}. The matrix $\mathbf{A}$ is computed based on the pair-wise inner products computed in $(\mathbf{X}\mathbf{W}_Q)(\mathbf{X}\mathbf{W}_K)^T = \mathbf{X}\mathbf{W}_Q\mathbf{W}_K^T\mathbf{X}^T$. Again we combine the inner matrices $\mathbf{W}_Q\mathbf{W}_K^T = \mathbf{W}_{QK} \in \mathbb{R}^{d_{\text{model}}\times d_{\text{model}}}$ which also has rank d_{head}. This matrix defines a bilinear form $\langle \mathbf{x}_1, \mathbf{x}_2\rangle = \mathbf{x}_1\mathbf{W}_{QK}\mathbf{x}_2^T \in \mathbb{R}^{d_{\text{model}}} \times \mathbb{R}^{d_{\text{model}}} \to \mathbb{R}$ between residual stream vectors that tell us how much attention each residual stream vector wants to pay to every other residual stream vector. An important observation is that the OV and QK matrices are computed independently, implying that what information to move between positions and what positions to move information between happens independently. However, that's not to say that the head hasn't been trained to coordinate these two operations in a certain ways.

2.3.4 MLP layers

MLP layers read each of the residual stream vectors, does some processing to them, and then writes back some result, where no interactions are taking place between token positions. In contrast to the attention layers, the hidden layer of the MLP is usually much larger than the residual stream (often around 4 times larger). This means that the MLP layers can perform more complex processing than the attention layers. There's also some evidence that the MLP layers are responsible for storing facts and do so in a look-up table like manner [Meng et al., 2022].

Privileged basis

In contrast with the residual stream, the MLP layer privileges neuron directions when representing features in its internal activations. This is due to the component-wise GELU activation function. However, the neurons are still highly polysemantic due to the very large number of features it tries to represent, forcing it to use superposition.

2.3.5 Attention heads vs. MLP layers

In transformer mechinterp the MLP layers are often seen as a larger challenge to interpret compared with the attention layers since they perform complex non-linear processing on features in superposition, while the attention layers are more linear and separable into small heads that can be understood largely in terms of attention patterns and information movement, although they are still far from easy to interpret.

2.3.6 Paths & Circuits

The residual stream can be decomposed into a sum of all the write operations that have been done to it, the first one being the embedding vectors. As an example let's imagine a one decoder block transformer. In this case the output logits are given by:

$$\mathbf{Z} = \mathbf{X}\mathbf{W}_E\mathbf{W}_U + \sum_{i}^{n_{\text{head}}} \mathbf{A}^i\mathbf{X}\mathbf{W}_E\mathbf{W}_{OV}^i\mathbf{W}_U + \text{GELU}(\mathbf{X}'\mathbf{W}_1)\mathbf{W}_2\mathbf{W}_U \tag{2.1}$$

$$\mathbf{X}' = \mathbf{X}\mathbf{W}_E\mathbf{W}_U + \sum_{i}^{n_{\text{head}}} \mathbf{A}^i\mathbf{X}\mathbf{W}_E\mathbf{W}_{OV}^i\mathbf{W}_U \tag{2.2}$$

Which is a sum of $n_{\text{head}} + 2$ terms. This gives us a decomposition of the logits, which we can use to gain insight into what components are penultimately responsible for the model's predictions, and then work backwards from there.

Another way to view it is that each term in the sum is a set of paths from the input to the output, for example $\mathbf{X}\mathbf{W}_E\mathbf{W}_U$ gives the direct paths from input embeddings to output logits along the same token positions, while $\mathbf{A}^1\mathbf{X}\mathbf{W}_E\mathbf{W}_{OV}^1\mathbf{W}_U$ gives the paths going through the first attention head, now potentially visiting different token positions. Not all paths are independent of course, for example the MLP's output is potentially dependent on any operation done to the residual stream before it. Or if we had more decoder blocks then the attention heads in later layers could depend on operations before. When a path goes through only two heads with no MLP interfering in between, it is equivalent to

$$\mathbf{A}^b\mathbf{A}^a\mathbf{X}\mathbf{W}_E\mathbf{W}_{OV}^a\mathbf{W}_{OV}^b\mathbf{W}_U \tag{2.3}$$

Where a, b denotes heads in different layers. In the transformer's computational graph, this could be seen as a path going from input embedding to head a to head b and then to output logits.

If we take the transformer's entire computational graph where nodes are embedding, individual heads, MLP layers, and unembedding, we can imagine individual paths going from input to output. More generally, a subgraph of this computational graph, potentially consisting of many paths, is called a **circuit**. A central theme in mechinterp is trying to find circuits in transformers that are performing a specific task, and then interpreting how exactly each node in the circuit comes together to perform the task.

2.3.7 Head composition

In equation 2.3 we imagined a computational path going through two heads, but there are many ways to compose heads in a transformer. First of all, they might not compose at all if they interact with orthogonal subspaces, which is not uncommon due to the low rank. If they do compose this could happen in one of three ways depending on which of the matrices $\mathbf{W}_K, \mathbf{W}_Q, \mathbf{W}_V$ are reading from the subspace written to by a previous head, respectively referred to as K-composition, Q-composition, or V-composition.

The type of composition can tell us something about what the head is doing. For example, a head doing K-composition is forming a key at a token position based on information written into that position by a previous head, this could for instance be which token comes before it: if you wanna predict what comes after a token, looking back in the context and to see what came after it before might be helpful, and now you have a position you can look up in the residual stream which contains this information.

2.3.8 Common types of heads

Empirically certain types of attention heads have been observed to consistently reoccur in transformers when they are trained to do next token prediction on text. I'll go through some of these now.

Previous token head

The previous token head performs a linear transformation on the positional embedding to pay attention to the token one position before, writing information about it into its own residual stream. This is a very typical head and is often observed in early layers. Due to the tokenization of the input it could be that the transformer would want to undo the tokenization in a way by looking at the previous token, combining them into a single representation. It's also used as building block for other more complicated heads like induction heads. It could work either by having a key be made at the source token that moves the positional embedding to match the one at the next position, or by having a query be made at the destination token that moves its positional embedding to match the one at the previous position, or some interpolation of the two. After the correct attention pattern has been made, the OV-circuit would take care of read/writing the information.

Duplicate token heads

Duplicate token heads look at previous instances of the token at a position and copies information about them, for example position, into its own residual stream.

Induction heads

Induction heads implement the following behavior: in a pattern like "A B ... A" predict that the next token is "B". Or in other words, given a context token, figure out what came after a previous occurrence of that token in the context window. For example in a sentence like "When John Johnson went to the store, John..." the induction head would move "Johnson" into the second instance of "John" so that can be offered as a prediction.

It has been observed [Elhage et al., 2021] that induction heads have been implemented in at least two different ways. One uses K-composition and one uses Q-composition.

- K-composition: The induction head keys are made by looking at the information copied in from a previous token head in an earlier layer. This allows the query to simply look for a previous instance of the current token in the subspace written to by the previous token head. The OV-circuit then copies over which token is at that position. This is the most common way that induction heads form.
- Q-composition: The induction head queries are made by rotating positional information copied in from a duplicate token head in an earlier layer one token position to the right. This allows the head to query directly into the right token position to find what token came after a previous occurrence.

[Olsson et al., 2022] did an in depth analysis of induction heads, arguing that the formation of induction heads is very common and that they're responsible for a majority of in-context learning for both large and small transformer language models. They've also found that induction heads in large models can do more fuzzy type of completion, performing a sort of analogical reasoning "A B ... A* $\rightarrow$ B*" where the relationship between A and B is comparable to the relationship between A* and B* in some embedding space, for example the same words in different languages.

2.3.9 Examples of circuits

The IOI circuit

[Wang et al., 2022] identified a circuit in GPT2-small that does indirect object identification (IOI). The IOI task is the following: given a sentence like "John and Mary went to the shop, then John gave a book to" predict that the next token is "Mary". Here "Mary" is the indirect object, "John" is the subject, "gave" is the verb, and "book" is the direct object. The general structure would look like "S1 and IO went to the shop, then S2 gave the book END" where "S1" and "S2" refer the the first and second instances of the subject respectively, "IO" is the indirect object, and "END" is the last token where the prediction happens.

The circuit implements the following algorithm:

1. Identify the previous names.
2. Remove the name that is duplicated.
3. Output the remaining name.

The main functionality of this algorithm is implemented using three types of heads (although this is a bit of a simplification and the one outlined in the paper is a bit more messy).

1. Duplicate token head: Usually it has each destination token pay attention to the beginning of the sentence (to the EOS token), this can be imagined as the resting state of the head. Otherwise it activates when it sees previous instances of the same token, particularly if the destination token is a name. In this case S2 would pay attention to S1 indicating the a duplication has occurred.
2. S-inhibition head: It moves information about the duplicated token into the "END" position, specifically into the subspace read by the query matrix of the name mover head, making it ignore the duplicated token.
3. Name mover head: moves previous names into the END position where the prediction happens, except the names that have been inhibited by the S-inhibition head.

The existence of this circuit is argued for by doing various ablation experiments, looking at how heads compose, and looking at attention patterns.

2.4 Evals

We care about figuring out how the transformer performs tasks. So first we need to actually measure whether it's doing a task. This is the point of evals. It involves validation data where it successfully and unsuccessfully performs the task And it involves a metric we can use to measure how well it's doing on the task. In this section I'll give an overview of evals in the context of IOI.

2.4.1 Making eval data

After identifying a task we need to find nice and clean data that we can feed the model that checks for the specific task. In the case of IOI we could have examples like

- "John and Mary went to the shop, then John gave a bag to" $\rightarrow$ " Mary".
- "Tom was riding a bike with Carl, then Tom crashed into" $\rightarrow$ " Carl".
- "Mary and John were sleeping, after Mary woke up she also woke up" $\rightarrow$ " John".
- "Mary, Carl and Tom were playing a game, then Carl and Tom teamed up against" $\rightarrow$ " Mary".

2.4.2 Metrics

A metric is just a measure of how well the model is performing a certain task. I'll go through a few possible measures for the IOI task.

Logits

Perhaps the simplest metric is the logit magnitude of the correct answer. However, this metric is problematic in a few ways. It is sensitive to a bunch of task independent factors like unigram statistics of the tokens: if a token is just more frequent it will tend to have a larger logit magnitude even though the there's nothing really about the context that should make it a more likely answer to the task. In addition, the model is not trained to optimize logits directly, but rather minimize cross-entropy loss which is equivalent to the average log-probabilities of the correct tokens. In fact, adding a constant to every logit doesn't change the log-probabilities since the softmax is invariant to shifting along the vector of 1's:

$$\frac{\exp(z_i + c)}{\sum_j \exp(z_j + c)} = \frac{\exp(z_i)\exp(c)}{\sum_j \exp(z_j)\exp(c)} = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

A benefit of logits though is that they are in some sense more interpretable than for example log-probabilities, given that they're what's directly outputted by the inner products of the residual stream with the unembedding matrix.

Log-Probability

Given the issues with logits, more reasonable metric might be log-probabilities. However it still suffers from the same problem of being sensitive to unigram statistics of the tokens and other types of task independent effects influencing the model's performance on the task. Also we lose the benefits of being more interpretable outputs of the model.

Logit difference

In the IOI task we want to isolate a very a specific behavior, namely how much more likely the model is to give the indirect object vs. the subject as the correct name prediction. This is because those are the 2 names that are relevant to the in-context behavior of the model. This motivates using the difference between the log-probabilities of the IO and S tokens as a metric. Doing this and running the prompts for all permutations of names also cancels out the task independent effects mentioned earlier, leaving us only with how much more likely the correct name is given the context. Taking a difference also allows us to use logits instead of log-probabilities, since the constant shift invariance mentioned earlier doesn't apply to differences due to the constant canceling out. Therefore logit difference is the standard metric used in the IOI task, which is what I'll also use.

2.5 The Mechinterp Toolkit

So assume we have our evals set up. Now we can start trying to understand how the transformer performs the task, or why it's not able to perform it if it can't. In this section I give an overview of various methods that can be used to mechanistically understand transformer models.

2.5.1 Max activation examples

A way to understand what a neuron is doing is to look at the dataset examples that maximally activate it. This can give us insight into the types of inputs that drive the neuron's response and help us understand the features it is sensitive to. We can also do this for any other property of the network, for example directions in the residual stream, or attention patterns.

2.5.2 Inspecting attention patterns

Plotting the attention patterns of the different heads on various inputs can give us insight into what purpose they serve in performing various tasks. Examples of things we can see in attention patterns are heads which only pay attention to the token directly before, or heads that pay attention to previous instances of the same token. A limitation with this approach comes from the fact that the transformer is trying to produce next-token predictions for all token positions at the same time, so in the case where we're analyzing a specific prediction at a specific token it can be difficult to differentiate information movement that has downstream relevance for that token vs. next-token predictions at other tokens. It also doesn't directly tell us anything about what information is being moved.

2.5.3 Direct Logit Attribution

Direct logit attribution is a technique used to analyze how different components of a neural network directly contribute to its final output logits. Direct logit attributions can be obtained using the decomposition in 2.1. This expansion allows us to attribute the contribution of each component to the final logits. For instance, the term XW_EW_U represents the direct path from input embeddings to output logits. These are the endpoints of the computational graph, giving us a starting point we can work backwards from.

2.5.4 Logit Lens

Logit lens is a technique introduced by [nostalgebraist, 2020] which unembeds the residual stream after each layer (MLP or attention), revealing how prediction confidence evolves across computational stages, leveraging the fact that transformers tend to build their predictions iteratively across layers. It provides insights into how different layers contribute to the final prediction and at what point in the network certain predictions become dominant. Unembedding the residual stream at any specific layer is essentially equivalent to deleting all subsequent layers.

2.5.5 Ablation

Given a the networks computational graph we can interfere with it by removing certain components and see if it affects the model's performance. This is known as ablation. Completely removing a component is equivalent to setting its activation to 0. Other types of ablation include setting the activation to a constant value, random value, or mean activation over some reference dataset. Ablations allow us to measure not just the direct contribution of components to the residual stream, but also downstream effects.

2.5.6 Activation Patching

Activation patching is a type of ablation strategy introduced in Meng et al. [2022]. The method works by swapping activations from a corrupted input with activations from a clean input or vice versa. The idea being that the corrupted input is chosen such that it's minimally different from the clean input, only being corrupted in a way that destroys performance on the particular task being investigated. A common motivation given for this approach is that other types of ablation takes the model too far out of distribution, causing unpredictable behavior [Chan et al., 2022]. The general procedure for activation patching is (in the case of corrupt to clean patching):

1. Run the model with a clean input and cache the activations.
2. Run the model with a corrupted input and store output.
3. Re-run the model with the corrupted input, but replace specific activations with those from the clean cache.
4. Observe changes in the model's output to determine the importance of the replaced components.

2.5.7 Attribution Patching

Attribution patching is a method that uses the gradient of the patch metric with respect to the activations to linearly approximate activation patching values. Given the large computational cost of having to do forward passes for every combinations of model components we want to patch, attribution patching can significantly speed up the process, at the cost of some accuracy. Attribution patching tends to work best on smaller components of the model, like heads, and worse on large components like the residual stream.

2.5.8 Automated Circuit Discovery

Some methods for doing automated circuit discovery include [Conmy et al., 2023] and [Syed et al., 2023]. They include various techniques like doing a series of activation patching runs, starting from the leaves of the computational graph and working backwards, to using gradient descent to learn masks over of activations in the computational graph that interpolates between the clean and corrupted activations.

2.5.9 Linear Probing

Linear probing involves training a linear classifier on network activations to see if the presence of certain feature can be detected. For example the network could be fed verb sentences either in past or present tense, then a linear classifier could be trained on various activations to see if this can be predicted. The success of a linear probe would be evidence in favor of the linear representation hypothesis. The weights of the linear probe could also be used to find feature directions: if the feature activation is given as the inner product

$$\mathbf{w}^T\mathbf{a} = f$$

then $\mathbf{w}$ is the direction of the feature. Some issues with linear probing is that it's supervised so we have to specifically decide which features to look for and construct a labelled dataset. It's also not guaranteed that the feature is actually being represented in the activations, and we're perhaps just finding a direction that's correlated with some set of the features that are actually being represented.

2.5.10 Sparse Autoencoders

It's known that transformers tend to represent features in superposition, making it difficult to identify which features are active given a specific activation vector. If there are more features than neurons then the feature basis is overcomplete and there are many ways to represent the activation vector as a combination of features, making it difficult to know which features are active. A way to get around this issue is to utilize the fact that features tend to be sparse. This creates a sparse dictionary learning problem. [Bricken et al., 2023] solves the dictionary learning problem using sparse autoencoders (SAEs), finding a large number of interpretable features in a one-layer transformer. [Templeton, 2024] uses the same technique on Claude 3 Sonnet, finding a large number of abstract and interpretable features. Much of the current research is centered on both figuring out how to scale SAEs up to even larger models, and how to use them to gain deeper insights, for example by finding interpretable circuits of features that explain how the model is "thinking" when making predictions. Some interesting examples of features identified in Claude 3 Sonnet include.

- Features for type signatures in code, specific people, specific locations.
- Features that are language independent, firing on the same concepts in different languages.
- Features that are multi-modal, firing on the same concept in both text and image input.
- Features that are fire on both concrete and abstract instances of a concept, like both code with security vulnerabilities and discussion of security vulnerabilities.

The typical setup for an SAE is to have an MLP with a single hidden layer try to reconstruct its input, where the hidden layer is larger than the input layer and has a sparsity penalty applied to it, typically L1-loss applied to the hidden activation vector.

If $\mathbf{x}$ is the input vector then an example of an SAE is given as

$$\begin{aligned} \mathbf{a} &= \text{ReLU}(\mathbf{W}_{\text{enc}}\mathbf{x} + \mathbf{b}) \\ \hat{\mathbf{x}} &= \mathbf{W}_{\text{dec}}\mathbf{a} \end{aligned}$$

Where the columns of $\mathbf{W}_{\text{dec}}$ contain the feature directions we're trying to learn and $\mathbf{a}$ contain the corresponding feature activations. The loss function might be given as

$$\mathcal{L} = \text{MSE}(\mathbf{x}, \hat{\mathbf{x}}) + \lambda \sum_i |a_i|$$

Where λ is the sparsity penalty hyperparameter. The SAE then tries to balance reconstruction loss and sparsity loss, the hope being that it will indentify the "true" sparse set of features present in a given activation vector.

However this type of SAE suffers from an issue called "shrinkage" where feature activations are systematically underestimated. This is since minimizing absolute value of feature activations creates a preference for smaller activations. This issue is solved by [Rajamanoharan et al., 2024] in an architecture called "gated SAE" by decomposing the magnitude and direction of the feature vectors, applying L1 only to the direction, achieving significantly better reconstruction and sparsity performance compared with the standard SAE.

2.5.11 Feature Attribution

Feature attribution ([Olah et al., 2018]) is a technique that allows us to see how features affect the output. Once a feature direction in the model have been identified using for example an SAE, we can take the directional derivative of a performance metric with respect to the feature direction to get an idea of how important the feature is for the model's performance on a certain set of inputs.

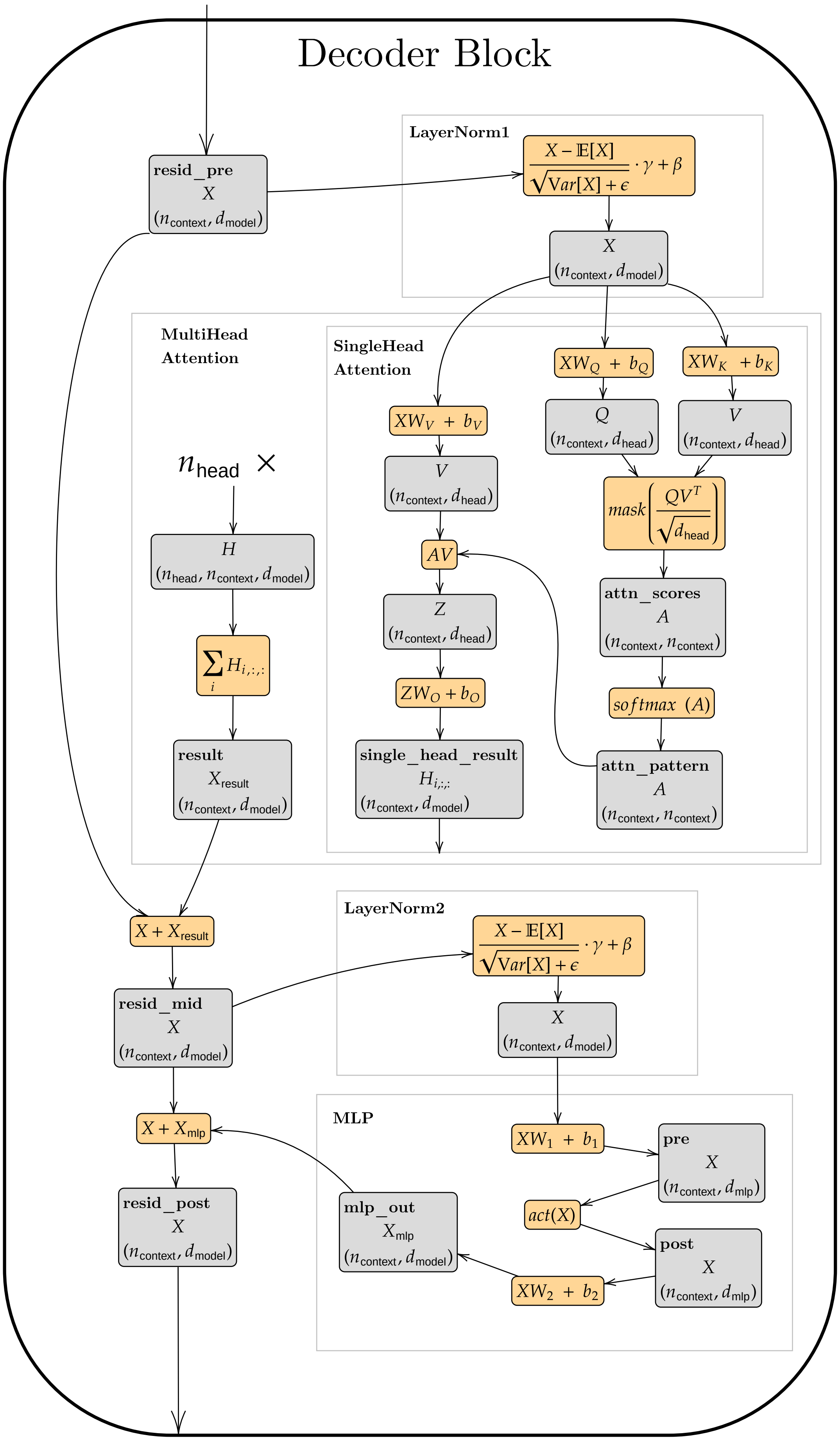
Decoder Block
resid_pre X $(n_{\text{context}}, d_{\text{model}})$
LayerNorm1
$\frac{X - \mathbb{E}[X]}{\sqrt{Var[X]+\epsilon}} \cdot \gamma + \beta$
X $(n_{\text{context}}, d_{\text{model}})$
MultiHead Attention
SingleHead Attention
$XW_Q + b_Q$
$XW_K + b_K$
$XW_V + b_V$
Q $(n_{\text{context}}, d_{\text{head}})$
V $(n_{\text{context}}, d_{\text{head}})$
$n_{\text{head}} \times$
V $(n_{\text{context}}, d_{\text{head}})$
$mask\left(\frac{QV^T}{\sqrt{d_{\text{head}}}}\right)$
H $(n_{\text{head}}, n_{\text{context}}, d_{\text{model}})$
AV
Z $(n_{\text{context}}, d_{\text{head}})$
attn_scores A $(n_{\text{context}}, n_{\text{context}})$
$\sum_i H_{i,:,:}$
$ZW_O + b_O$
$softmax\ (A)$
result X_{result} $(n_{\text{context}}, d_{\text{model}})$
single_head_result $H_{i,:,:}$ $(n_{\text{context}}, d_{\text{model}})$
attn_pattern A $(n_{\text{context}}, n_{\text{context}})$
$X + X_{\text{result}}$
LayerNorm2
$\frac{X - \mathbb{E}[X]}{\sqrt{Var[X]+\epsilon}} \cdot \gamma + \beta$
resid_mid X $(n_{\text{context}}, d_{\text{model}})$
X $(n_{\text{context}}, d_{\text{model}})$
$X + X_{\text{mlp}}$
MLP
$XW_1 + b_1$
pre X $(n_{\text{context}}, d_{\text{mlp}})$
$act(X)$
resid_post X $(n_{\text{context}}, d_{\text{model}})$
mlp_out X_{mlp} $(n_{\text{context}}, d_{\text{model}})$
post X $(n_{\text{context}}, d_{\text{mlp}})$
$XW_2 + b_2$

Figure 2.1: The decoder block's computational graph. Orange boxes are functions and gray boxes are tensors with dimensions specified.

Experiments & Discussion

3.1 Introduction

In the first part I'll redo some of the analysis that was done in [Wang et al., 2022] for both GPT2-small and tiny-stories-8M to see what the similarities and differences are in how they solve the IOI task. In the second part I'll train and apply a gated sparse-autoencoder on tiny-stories-8M with the purpose of figuring out to what extent various features are used in the model to solve the task.

3.2 Models

In this section I give some brief background on the language models I'll be investigating. All of the models used are publicly available on `huggingface.co`.

TinyStories-8M

[Eldan and Li, 2023] introduced a family of decoder only transformer language models trained on the TinyStories dataset: a small dataset of synthetic short stories using very simple language intended to be understandable to 4-year-olds. I'll specifically use the 8M million parameter one as this is the smallest one in the family that was able to perform the IOI task.

GPT2-small

[Radford et al., 2019] introduced a family of decoder only transformer language models trained on the WebText dataset: a large dataset of millions of webpages. I'll be looking at the smallest one in the family (GPT2-small), which has 117M parameters.

Model Properties

Model	n_{params}	n_{layers}	d_{model}	n_{heads}	activation	n_{ctx}	d_{vocab}	d_{head}	d_{mlp}
tiny-stories-8M	8M	8	256	16	gelu	2048	50257	16	1024
gpt2-small	117M	12	768	12	gelu	1024	50257	64	3072

Table 3.1: Model architecture comparison

3.3 Experiment 1 (looking for common heads in TinyStories-8M)

I run some simple tests to see which of the heads in TinyStories-8M correspond to one of the commonly known classes of heads, specifically I look for previous token heads, induction heads, and duplicate token heads. These same tests are done in [Wang et al., 2022]. The tests work by scoring each head according to properties of its attention pattern on certain inputs. Specifically:

- **Previous token heads** The fraction of attention weight which is on the previous token position given sequences of random (uniformly sampled from dictionary) tokens.
- **Induction heads** Scores are produced as follows:
  1. Take a random sequence of tokens, then concatenate a copy of it to itself. For example if the random string is "abc", then the duplicated string is "abcabc".
  2. Now, since the tokens are randomly taken from the dictionary, the only way to predict the next token is to use some in-context mechanism, such as induction heads. Given a pattern like "AB ... A" induction heads pay attention to the B from the second instance of A. So, if the length of the sequence is $2n$, we can see to what extent the heads are attending from position i to $i - (n - 1)$, i.e. the one that comes after the previous instance of the current token.
  3. Now take the fraction of attention weight that is describe above as the score.
- **Duplicate token heads** The fraction of attention weight which is on the previous instance of the current token given sequences of random tokens.

I run these tests on both TinyStories-8M and GPT2-small and compare the results. The results for previous token scores are shown in figures 3.1 and 3.2. We can see that TinyStories-8M has far weaker previous token heads than GPT2-small, which has a head (H4.11) where approximately all the attention is on the previous token.

Previous token heads are needed to create induction heads that are formed using K-composition, which is the most common type of induction head. In figures 3.3 and 3.4 we can see the induction head scores of the two models. Remarkably, TinyStories-8M have no induction heads, while GPT2-small, as previously observed [Olsson et al., 2022], has many.

Finally, in figures 3.5 and 3.6 we can see the duplicate token scores. TinyStories-8M has a single weak duplicate token head head (H4.0), while GPT2-small has two strong duplicate token heads (H3.0) and (H0.5).

3.4 Experiment 2 (finding the IOI circuit in TinyStories-8M)

In this section I will go through my experiments analyzing how TinyStories-8M solves the IOI task and how it differs from GPT2-small and the results presented in [Wang et al., 2022]. I will discuss the results throughout this section.

3.4.1 Performance on IOI task

TinyStories-8M achieves an average logit difference of 9.4 (bigger is better) on the IOI task (difference of IO and S token logits over a batch of IOI prompts with names permuted), while GPT2-small achieves a logit difference of 3.6 on the same test set. This might be surprising given that TinyStories-8M is a much smaller model, but it can make sense if the model is more specialized for the task, which seems plausible given that the structure of the IOI task closely fits the type of text present in the TinyStories dataset.

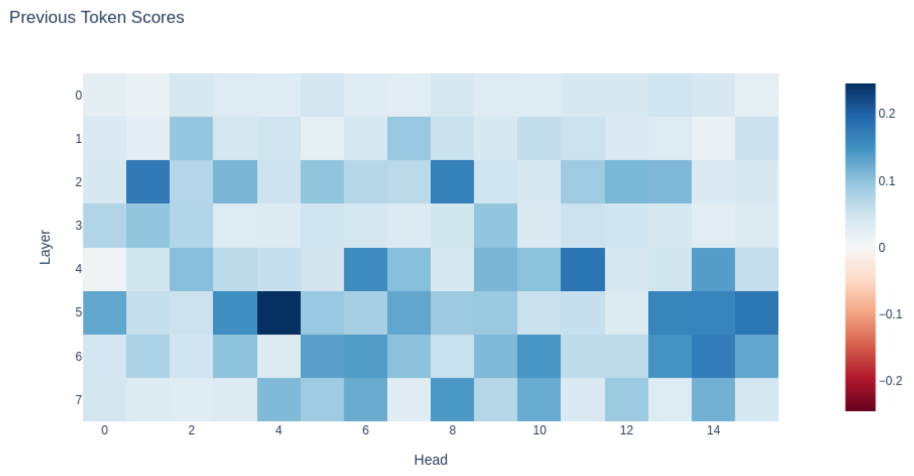

Figure 3.1: TinyStories-8M previous token scores.

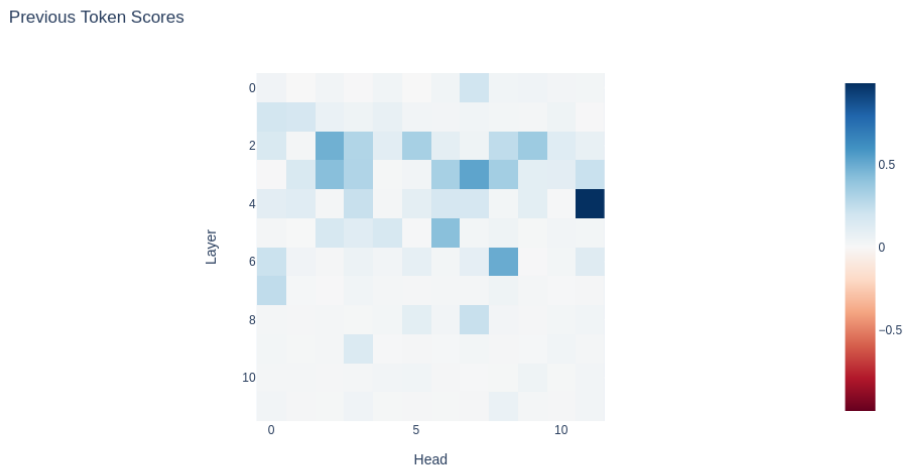

Figure 3.2: GPT2-small previous token scores.

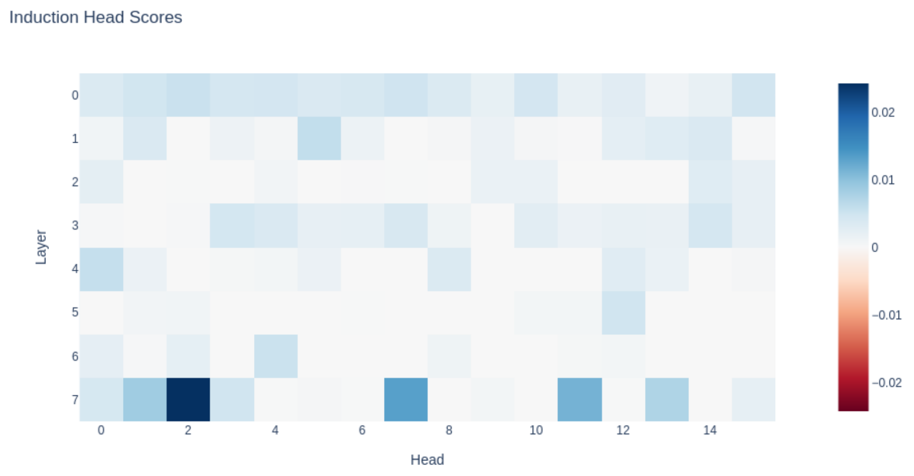

Figure 3.3: TinyStories-8M induction head scores.

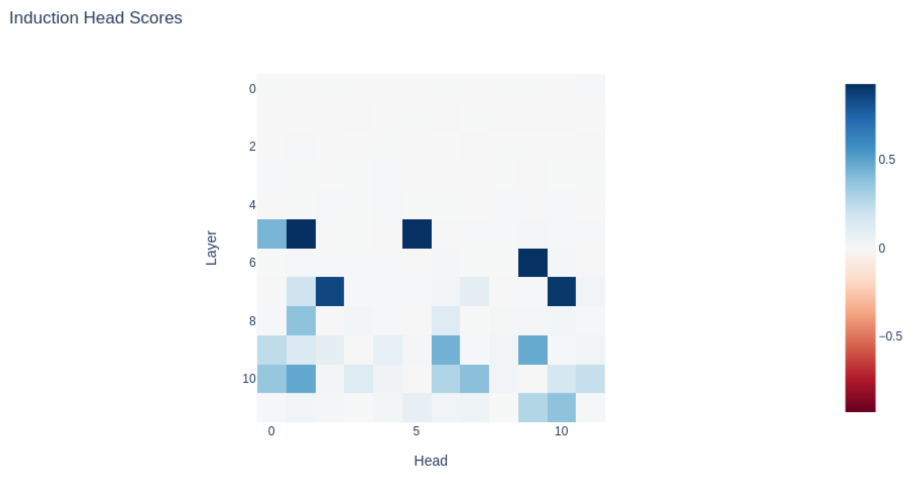

Figure 3.4: GPT2-small induction head scores.

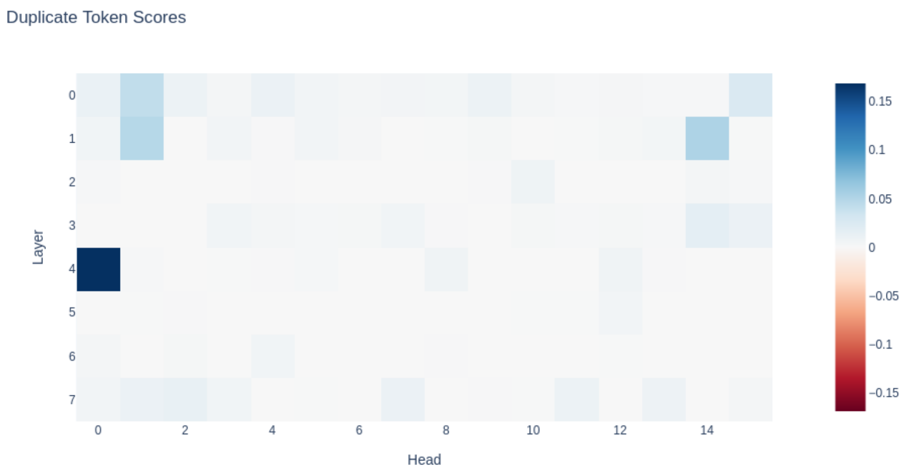

Figure 3.5: TinyStories-8M duplicate token scores.

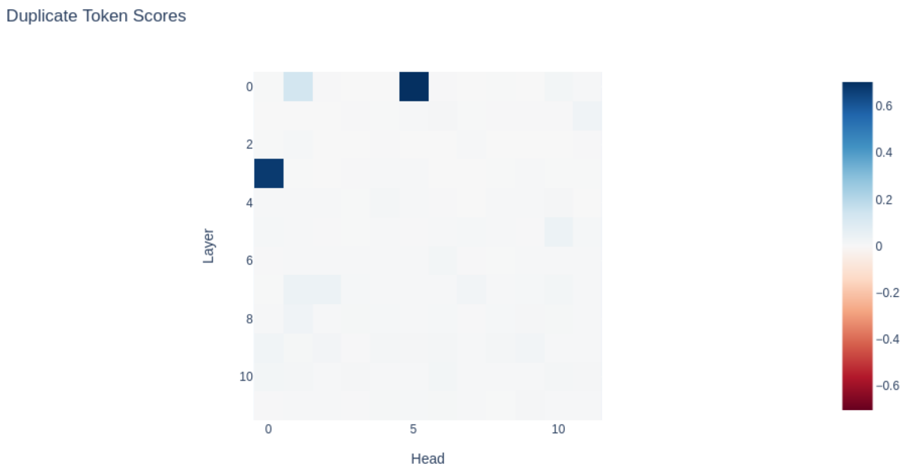

Figure 3.6: GPT2-small duplicate token scores.

3.4.2 Identifying the circuit

In figure 3.7 and 3.8 we can see the logit difference across the residual stream for TinyStories-8M and GPT2-small, respectively. This is equivalent to decoding the residual stream after each layer, seeing to what extent the right answer has already been written into the stream.

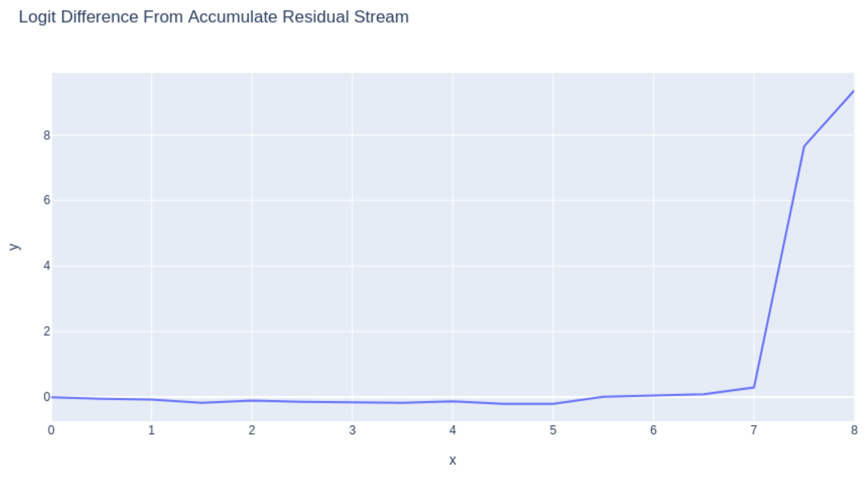

Figure 3.7: TinyStories-8M logit difference across residual stream. Performance only increases. x is layer, y is logit difference. Integer ticks correspond to attention layers, while half integer ticks correspond to MLPs

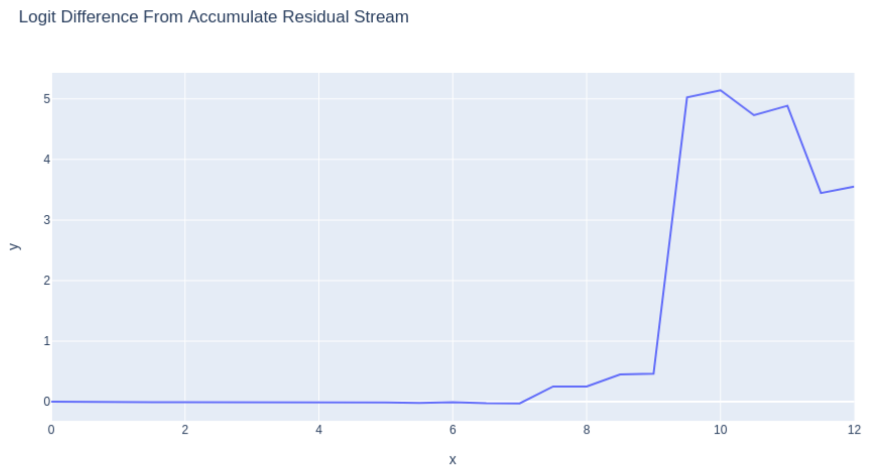

Figure 3.8: GPT2-small logit difference across residual stream. Performances sometimes decreases. x is layer, y is logit difference. Integer ticks correspond to attention layers, while half integer ticks correspond to MLPs

First we can notice the majority of direct logit attribution is achieved at a single layer in both models, meaning there's a head or MLP which confidently writes the correct answer into the residual stream. This occurs at layer 7 in TinyStories-8M and layer 9 in GPT2-small.

We can see that in TinyStories-8M the logit difference only increases across the residual stream, while in GPT2-small the logit difference sometimes decreases. The dip in performance is mentioned in [Wang et al., 2022] and is attributed to *Negative Name Mover Heads* which write in the opposite direction of

Name Mover Heads which writes the correct name from earlier in the context into the residual stream at the final position. They speculate the purpose of this head is to hedge against too confident predictions to lessen the impact of increased loss when making mistakes. The TinyStories distribution is far more structured and predictable than the WebText distribution, which might be part of the reason why negative name mover heads don't exist in the TinyStories-8M IOI circuit: given the lower variance in the plausible next tokens, dedicating capacity to learning a specific head for hedging might not be a worthwhile strategy. Another reason might be that TinyStories-8M has far fewer parameters, and so perhaps has less capacity to dedicate to more esoteric niche strategies. This last reason seems a bit implausible though given that it has about the same number of heads as GPT2-small, so it's not clear why it couldn't just dedicate a head to this if it had use for it.

Instead of decoding the residual stream, we can instead decode the vectors written to the residual stream by each layer. However, this would be equivalent to taking the difference between the residual stream at adjacent positions. What is more interesting is decoding the vectors written by individual heads. The result of this is displayed in figures 3.9 and 3.10. From these figures we can see that

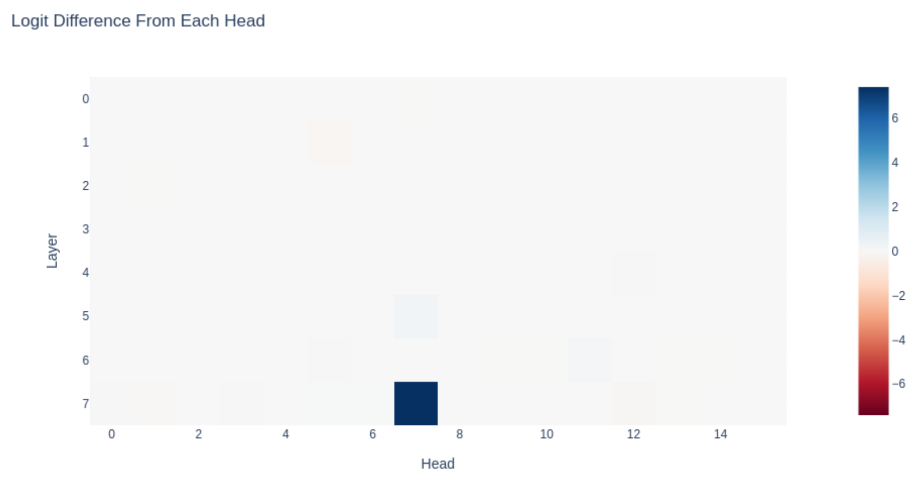

Figure 3.9: tinystories-8m logit difference from each of the heads write operations.

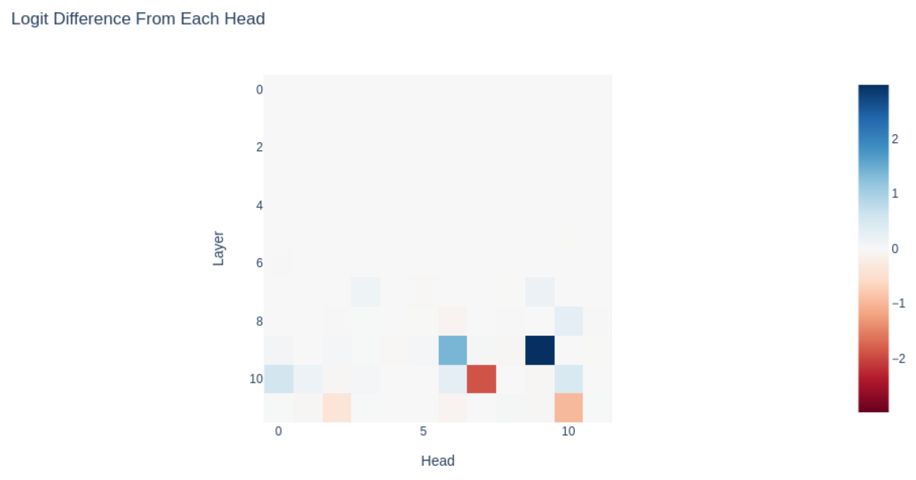

Figure 3.10: gpt2-small logit difference from each head write operations.

only a single head (H7.7) in the entire model has any noticeable direct impact on the logit difference in TinyStories-8M. While in GPT2-small the impact is more distributed, and there are negative name mover heads. This is additional evidence that TinyStories-8M doesn't have negative name mover heads.

Decoding the write operations and residual streams at the various layers gives useful information, but it's limited since it doesn't tell us anything about importance of components in the computational graph that are not at the leaves. To achieve this we can do activation patching on the intermediate components of the model and see how this impacts performance.

I will now present a series of corrupt-to-clean activation patching experiments where the clean sentence is "When John and Mary went to the shops, John gave the bag to" and the corrupted sentence is "When John and Mary went to the shops, Mary gave the bag to". In figures 3.11 and 3.12 we can see the result of patching the residual stream at each layer and token position.

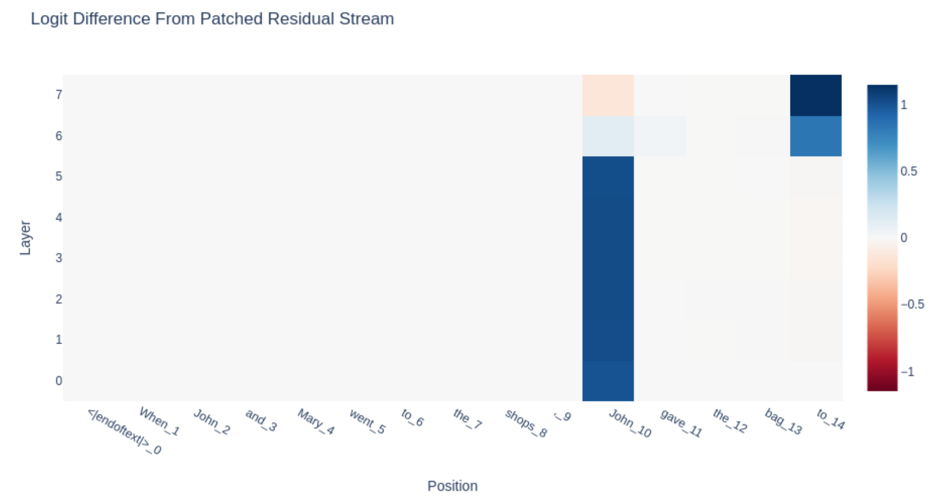

Figure 3.11: tinystories-8M logit difference from patched residual stream.

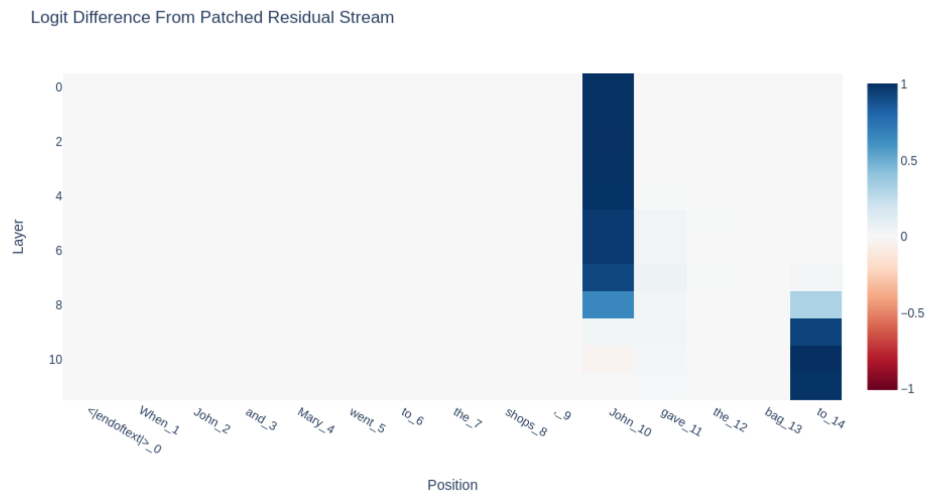

Figure 3.12: GPT2-small logit difference from patched residual stream.

From these plots we can see that, just like in GPT2-small, TinyStories-8M is using some information stored in the S2 position in one of the later layers to get the performance relevant information into the END position. We can also do activation patching on individual heads to get an understanding of not just immediate impact on the logit difference, but also how they matter as intermediate nodes in the computational graph. The result of this is shown in figures 3.13 and 3.14. In TinyStories-8M we still have that H7.7 matters quite a lot, but now we can also see that H5.7 is important, although not directly for the logits.

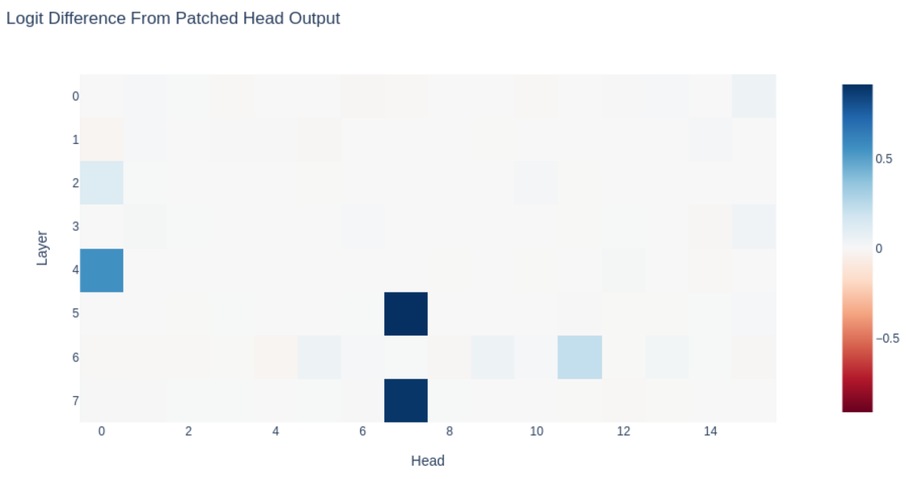

Figure 3.13: TinyStories-8M logit difference from patched head outputs.

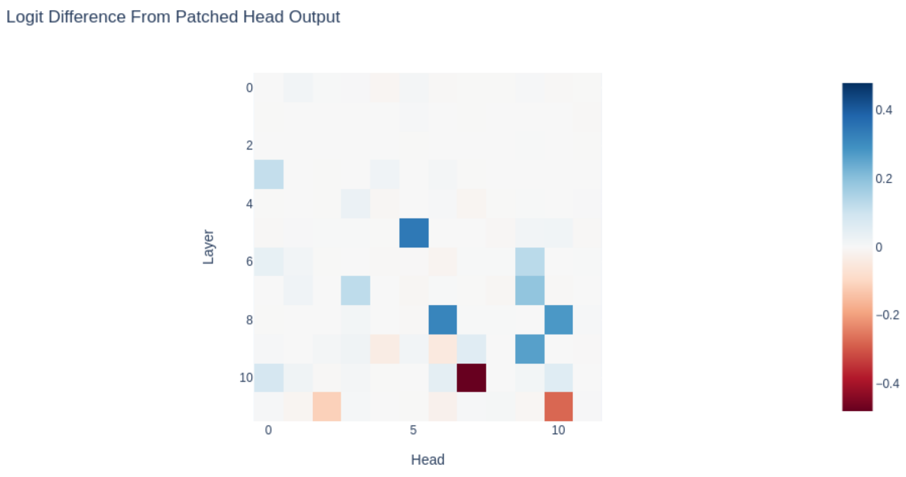

Figure 3.14: GPT2-small logit difference from patched head outputs.

In [Wang et al., 2022] H9.9, H9.6, and H10.0 are classified as the strongest name mover heads, they also have the largest logit difference attribution. The respective attention patterns of H9.9 and H9.6 can be seen in figures 3.15 and 3.16. Of notice is that they're quite similar on this input, however this doesn't necessarily mean they produce similar attention patterns in general.

It's now of interest to see if H7.7 in TinyStories-8M has a similar attention pattern to the name mover heads in GPT2-small, given that it was the only head with a significant direct attribution to the logit difference. In figure 3.17 we can see that the attention pattern of TinyStories-8M's H7.7 is quite similar to those of GPT2-small's H9.9 and H9.6.

In all of the attention patterns we can see that information is being moved from the IO position where the "Mary" token is located to the END position where the "to" token is located. We can also

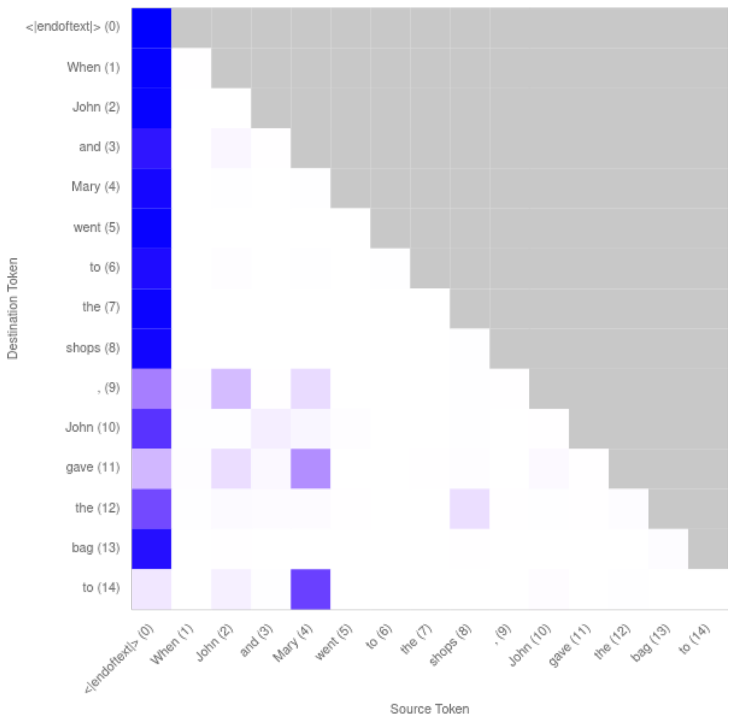

Figure 3.15: gpt2-small attention pattern for H9.9 on IOI input

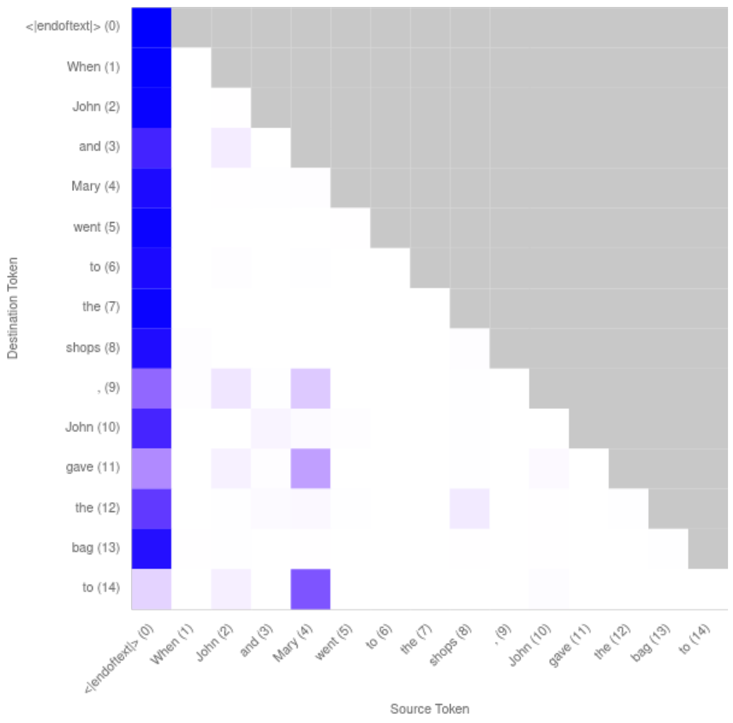

Figure 3.16: gpt2-small attention pattern for H9.6 on IOI input

see that IO information is being moved into the position where the token ”gave” is located. Presumably this is because ”Mary” is a reasonable prediction after ”gave” in the context of this sentence. This is something to keep in mind when interpreting attention patterns: the model is trying to predict the next token at all positions at the same time, so you have to differentiate the information that is important for the specific prediction you care about.

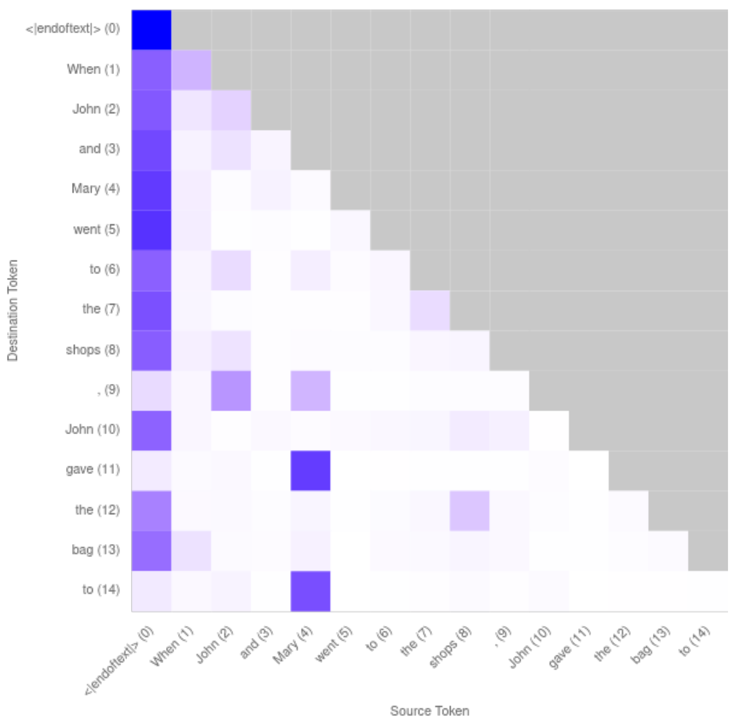

Figure 3.17: TinyStories-8M attention pattern for H7.7 on IOI input

Next, we’ll look at GPT2-small’s H8.6 in figure 3.18, which is the strongest S-inhibition head. Its job is to copy information from the S2 position into the END position so that the name mover head doesn’t copy the name corresponding to S2 into the END position. We can notice that it sends information from S2 into the END position and into the ”gave” position, but also from S1 into the following ”and” position. All places where predicting ”John” would’ve been unnatural.

In general, the probability of a name being the next token increases if it has appeared recently in the context. However, there are some cases where the name appearing previously in the context makes it less likely to occur again. This is where heads like the S-inhibition heads comes into play: it’s unnatural that John would give something to John, or that the sentence would contain parts like ”John and John”.

Other than TinyStories-8M’s H7.7 head, H5.7 also had a high head output patching score in figure 3.13. Now let’s look at the attention pattern of this H5.7 head in figure 3.19. We can see that it sends information from S2 to the end position, but also to the ”gave” position. Also, like GPT2-small’s H8.6, it sends information from S1 to the following ”and” position. It also didn’t have any direct contribution to the logit difference, and only had importance as an intermediary computation in the residual stream. These observations is an indication that H5.7 is a S-inhibition head comparable to the one in GPT2-small, although the activation pattern is not as clean as the one in GPT2-small.

The observations up to this point tells us that TinyStories-8M might have a similar IOI circuit to that of GPT2-small. However there are some differences: we can observe that the circuit is far simpler and more straightforward in TinyStories-8M compared with GPT2-small. There’s a smaller amount of heads used to perform the task, and there’s no negative name mover heads. In [Wang et al., 2022] they

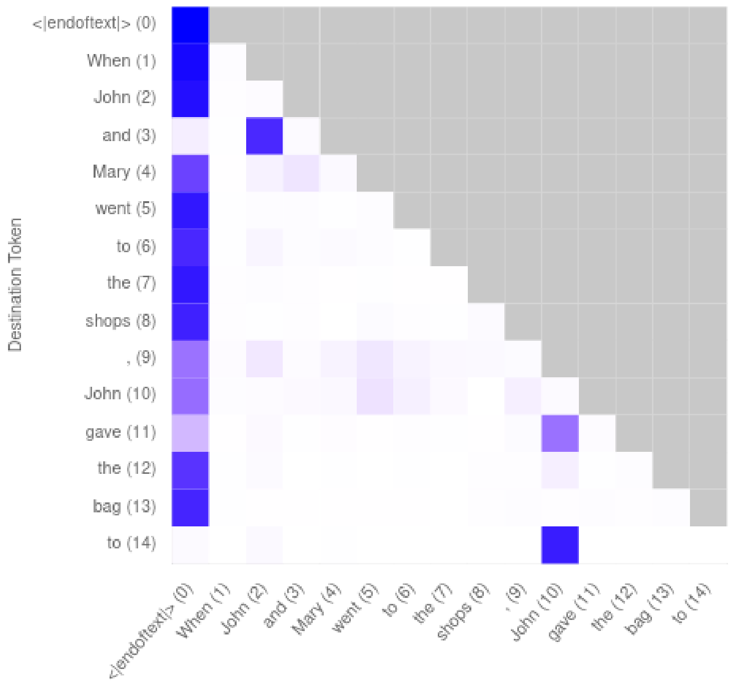

Figure 3.18: GPT2-small attention pattern for H8.6 on IOI input

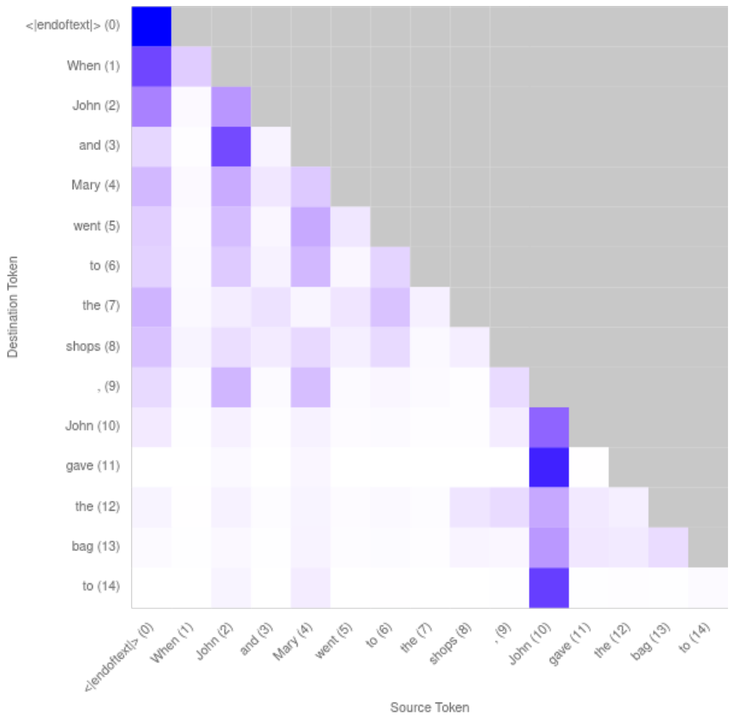

Figure 3.19: TinyStories-8M attention pattern for H5.7 on IOI input

also find that the GPT2-small IOI circuit reuses induction heads to function as duplicate token heads, but TinyStories-8M have no induction heads.

The IOI circuit in TinyStories-8M is then hypothesized to be:

1. Duplicate token head H4.0 copies duplicate name information from S1 to S2.
2. S-inhibition head H5.7 reacts to the fact that the name in S2 is a duplicate and sends this information to subsequent positions, especially ”gave” and ”to”.
3. Name mover head H7.7 copies the previous names into the correct subspace in the END position, except the ones that are inhibited by H5.7.
4. The name in the END position is then unembedded into a prediction.

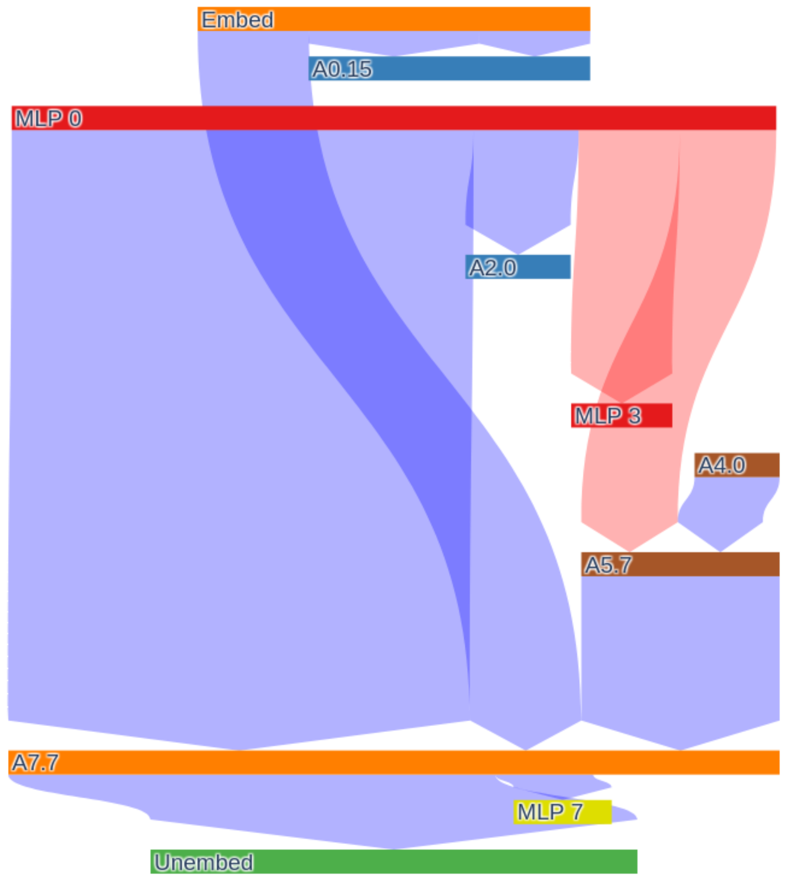

Figure 3.20: TinyStories-8M automatically discovered IOI circuit. Blue arrow means positive influence, red arrow means negative influence. Thickness of arrow corresponds to strength of influence. Ax.y means layer x’s yth attention head.

To offer additional evidence for this hypothesis I ran the edge attribution patching algorithm from [Syed et al., 2023] on the computational graph of TinyStories-8M, where nodes are: MLPs, embed/unembed values and key/query/value values. This produced the circuit shown in figure 3.20. And also gave me information on the types of composition taking place in the in-edge of each attention head node. Specifically:

- H7.7's value matrix reads from the output of MLP0 and Embed.
- H7.7's query matrix reads from the output of H5.7.
- H5.7's value matrix reads from the output of H4.0.

First of all, the fact that heads tend to read from MLP0 instead of the Embedding output is a common pattern in transformer language models and probably not specific to the IOI task (a typical theory is that it works as a sort of extended embedding).

How do these types of composition align with the previous story of how the circuit is functioning?

- S-inhibition head H5.7 needs to send duplicate name information (created by H4.0) to the correct subspace at the END position. This is achieved by V-composition, which is exactly what happens.
- Name mover head H7.7 needs to query for the positions of the non-duplicated names in the context, it can do this by having learned to query for embeddings that indicate a name, except the query matrix reads from the output of H5.7 to know which names to ignore.
- After knowing which names to pay attention to, H7.7 needs to copy these names into the correct subspace in the END position from the embedding/MLP0 output subspaces which is consistent with the results from the edge attribution patching.

In conclusion there seems to be a simple and clear circuit in TinyStories-8M that solves the IOI task. It's less messy than the one in GPT2-small, but otherwise identical in the core algorithm it's implementing, despite being much smaller and trained on a much simpler dataset.

3.5 Experiment 3 (Gated Sparse Autoencoder on TinyStories-8M)

In Experiment 2 we found the circuit in TinyStories-8M that implements IOI and created a mechanistic story for how it works that corresponds closely with the one in [Wang et al., 2022]. In this section we want to figure out what is semantically going on inside the model as this circuit runs. Specifically, what features are being used as part of this circuit to solve the task. To do this we train a gated sparse autoencoder ([Rajamanoharan et al., 2024]) on the residual streams at layer 6, 7, and 8. Then use the trained SAE to investigate the features used in the model. These layers in specific because there's not enough time to train and investigate SAEs for all layers, and these are the ones where the most important parts of the IOI circuit are located.

I will now go through each of the three layers, describing properties of features in that layer and how they relate to the IOI task.

3.5.1 Layer 5

First some general statistics for the feature activations in this layer.

By running the SAE on IOI prompts I found which features activated the most, the top 5 are listed in table 3.2.

However this only gives us which features were the most activated, that doesn't necessarily mean they were used for the specific task in question. So I also calculate the feature attribution scores for each feature with respect to the IOI task. The top 5 features in terms of feature attribution are listed in table 3.3

Feature	Activation
44	0.91
1276	0.50
3107	0.49
3638	0.37
3397	0.36

Table 3.2: Top activating features in layer 5 on IOI prompts.

Feature	Attribution
2338	5.15
2748	1.96
1725	1.82
4021	1.64
1214	1.54

Table 3.3: Features with highest feature attribution on IOI in layer 5

Once a few features have been identified as potentially important they can be manually investigated. As part of determining if they're relevant for the task I will employ an ablation technique where I project the residual stream in layer 5 onto the subspace orthogonal to the feature direction and see if this impacts IOI performance. Ablating random features doesn't impact performance by any significant amount.

Feature 44 (the highest activating one) seems to activate mostly at the ”to” token in the context of someone giving something to someone else. Ablating this feature changes correct name probability from 77.3% to 4.8%. In table 3.4 we can see the top activations of this feature in the TinyStories dataset.

Table 3.4: Max activating tokens with surrounding context for feature 44 in layer 5. Token position is bolded.

token	context
to	She takes out her money. She counts it. She has five dollars too. She gives it **to** the shopkeeper.
to	angry. He wants to play with the car. Ben picks up the blue car. He gives it **to** Mia. He says,
to	Can I have it, Tom? Can I have it? Tom nods and gives the ball **to** Lily. Lily hugs the
to	eat it! He digs out the cake and cuts a slice for Mia. He gives it **to** her and says, ”
to	face? He opened his bag and took out some buttons and a carrot. He gave them **to** Lily and Ben.

A common pattern is that features can either work to directly alter the logits produced by the model, or they can work as intermediary computational units. A feature activating at ”to” in the context of someone giving something to someone else would likely increase the logits corresponding to names or pronouns if it was a direct logit contribution feature, however multiplying this feature direction with the unembedding matrix we get that the largest logit contributions are for random words that have little to do with the IOI task. Something strange is that this feature had quite large negative feature attribution, in fact third most negative of all features, meaning that increasing it locally decreases logit difference. By inspecting the attention patterns in layer 6 I found that H6.11 is also a name mover head, which is consistent with the head patching in 3.13, although a weaker one than H7.7. It seems that feature 44 is somehow related to this head's ability to move previous names in the context into the ”to” position, perhaps just by working as a signal that it should do it. Ablating this feature modifies the attention pattern in H6.11 to have ”to” not attend to previous names anymore, but otherwise leaves it mostly unchanged. See figures 3.22 and 3.21 for the difference in the attention before and after ablating. Instead attention is redirected to the comma ”,” and the end of text token, both of which tend to act as resting positions for attention in transformer language models.

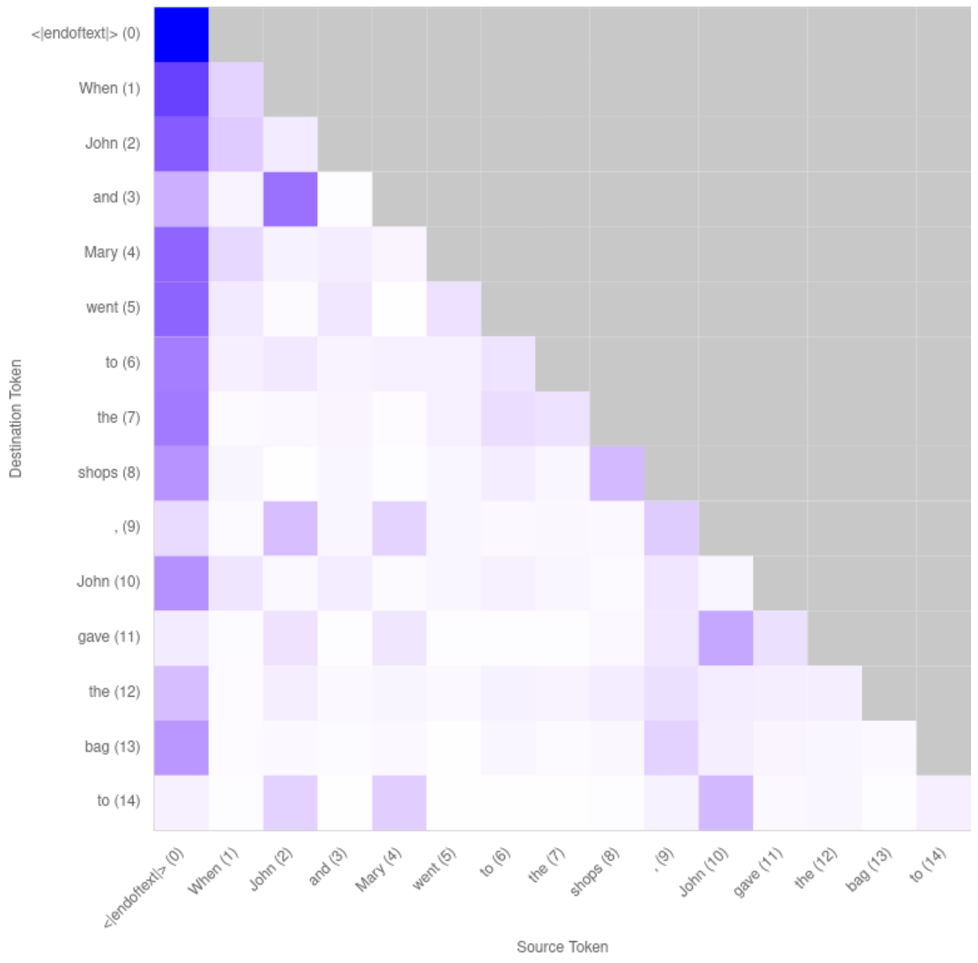

Figure 3.21: L6H11 attention pattern when **not** ablating feature 44 in layer 5. We can see that names are being moved to the ”to” position.

It also influences the attention pattern of H7.7, making it that the attention from ”to” to IO changes from 0.7 to 0.3, although it is still the only position attended to except for the beginning of string resting position.

In contrast to H7.7 which only moves the IO name 3.17, H6.11 moves both the subject and the indirect object into the final ”to” position, indicating that there’s no corresponding S-inhibition head that synergizes with it. This might explain the large negative attribution score for feature 44: decreasing it will make it so H6.11 stops moving the wrong name into the final token position, which would decrease the score according to the logit difference metric.

Some other features of note are feature 1276 which activates at ”to” in the context of infinitive verbs like ”to play” or ”to eat”. What’s weird about this feature is that ablating it made literally every head in every subsequent layer pay strong attention to previous instances of ”to”. An example is shown in figure 3.23, but all the heads in all subsequent layer looked almost identical. I was thinking this maybe emulated the resting position property of beginning of string token, but inspecting inner product of the query vector made from the beginning of string token and query vector made from the ”to” token showed that they were no closer than any other two random tokens.

Feature 2338 has significantly higher attribution scorer than the others and activates on ”gave” in

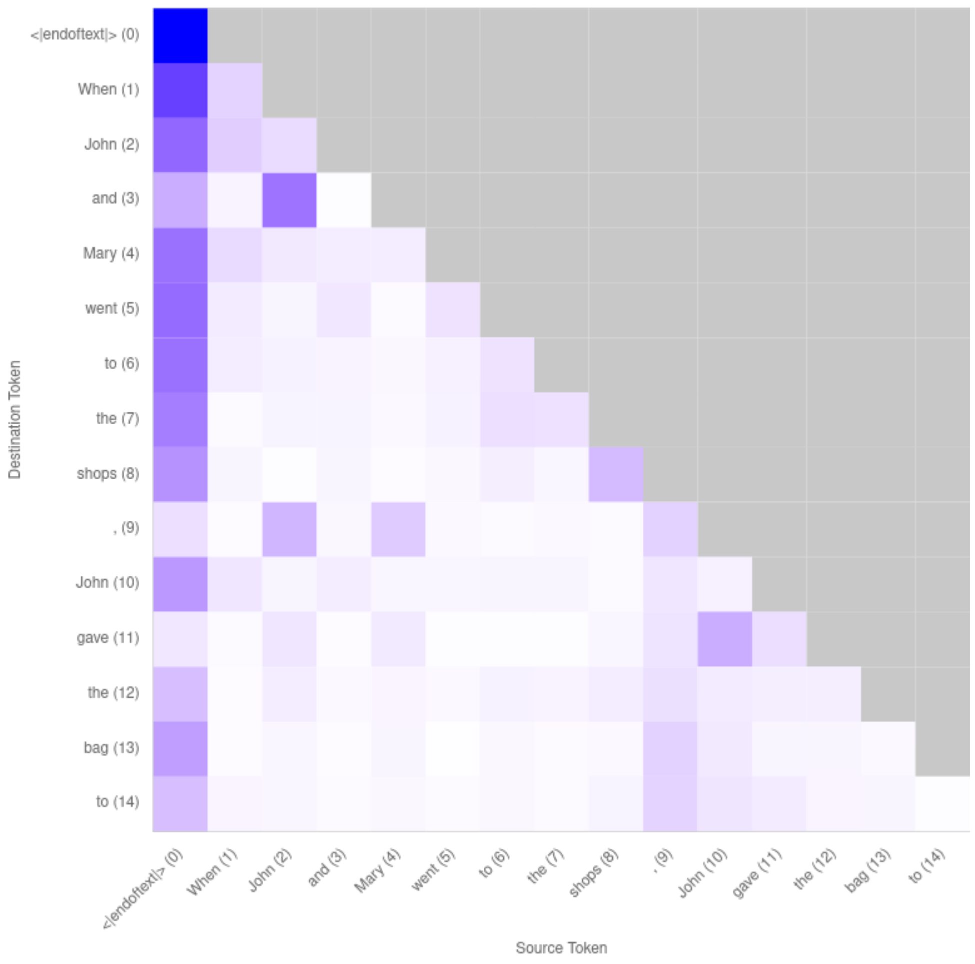

Figure 3.22: L6H11 attention pattern when ablating feature 44 in layer 5.

the context of someone giving something to someone else. Ablating it changes the attention paid by H7.7 from ”to” to IO from 0.7 to 0.3. Although this influence has to be a bit more indirect since it activates at the ”gave” position but influences the attention between the ”to” and IO positions 2 layers later. Furthermore it doesn’t influence the attention pattern of H6.11 significantly. One possibility is that this feature gives context that is used to interpret the subsequent ”to” token as someone giving something to someone else.

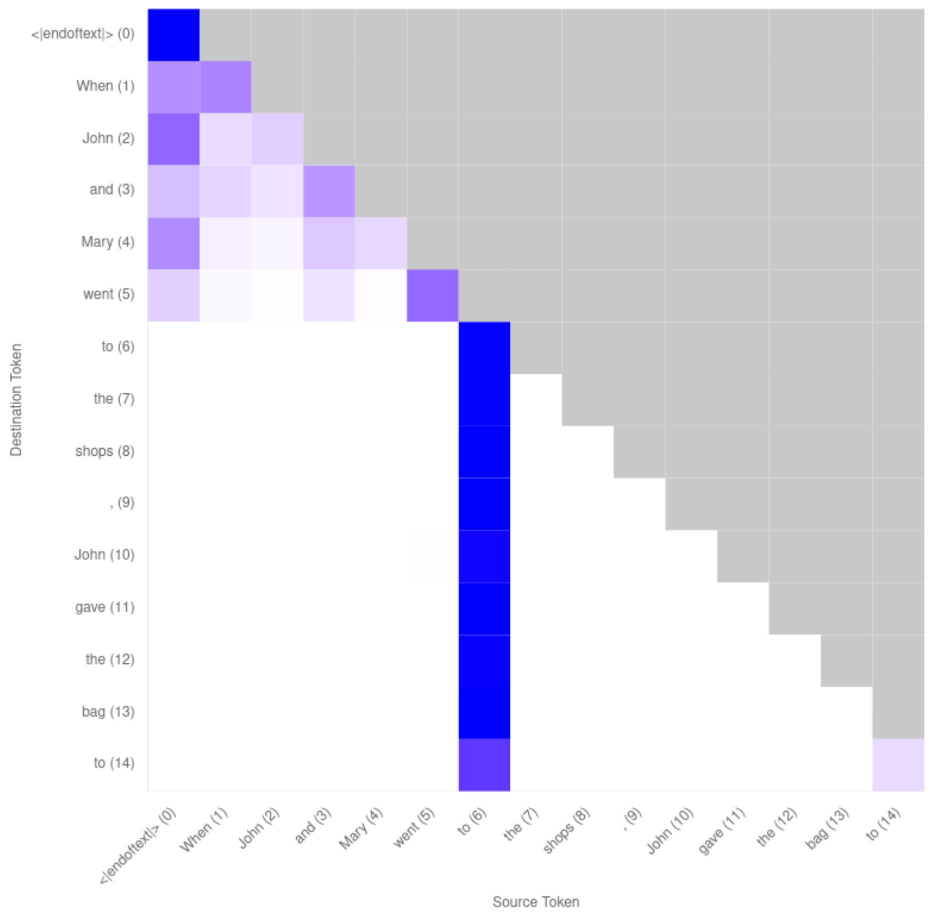

Figure 3.23: Weird attention pattern after ablating feature 1276 in layer 5.

3.5.2 Layer 6

In layer 6 the top activating features and those with highest feature attribution for IOI are listed in tables 3.5 and 3.6 respectively.

Feature	Activation
1059	0.77
41	0.50
3176	0.31
1156	0.27
1237	0.23

Table 3.5: Top activating features in layer 6 on IOI prompts.

The most activating feature in layer 6 is feature 1059, which activates on ”to” in the context of someone giving something to someone else (see table 3.7 for top activating examples). Ablating it

Feature	Attribution
41	4.70
2184	3.32
964	2.30
2837	2.07
1986	1.72

Table 3.6: Features with highest feature attribution on IOI in layer 6

changes the correct name probability from 77.3% to 0.25%. Ablating it also makes the name mover head H7.7 no longer pay attention to the previous names in the context. So it seems like the relationship of this feature to H7.7 is the same as the one of feature 44 to H6.11.

Table 3.7: Max activating tokens with surrounding context for feature 1059 in layer 6. Token position is bolded.

token	context
to	angry. He wants to play with the car. Ben picks up the blue car. He gives it **to** Mia. He says,
to	She takes out her money. She counts it. She has five dollars too. She gives it **to** the shopkeeper. The
to	face?" He opened his bag and took out some buttons and a carrot. He gave them **to** Lily and Ben.
to	barks. It nods its head. Ben breaks the biscuit in half. He gives one half **to** the dog. He eats
to	Can I have it, Tom? Can I have it?" Tom nods and gives the ball **to** Lily. Lily hugs the

In figures 3.24 and 3.25 you can see the attention patterns of H7.7 before and after ablating feature 1059. The effect is identical to the one for feature 44 to H6.11 in figure 3.22, in contrast this head pays attention only to the correct name, making it clear that this feature is not relevant to the mechanism that allows the head to pay attention only to the correct name.

Feature 41 has both high activation and attribution score, but the top activating examples do not paint a clear picture of when exactly it activates, it seems to be something to do with verbs of various tense and type. Also multiplying the corresponding feature direction with the unembedding matrix don't give any clear logit contributions either.

Feature 2837 has by far the largest negative attribution score (see table 3.8). Looking at the top activating examples in table 3.9 for this feature it seems like it activates on a cluster of names, one of which is the name "Mary", but I also noticed the "John" appears in one of the later examples. Regardless it is not clear exactly why this feature has such negative large attribution. Strangely enough, ablating it makes all previous heads pay attention to only the previous names in the context.

Feature	Attribution
2837	-7.79
1507	-3.93
1182	-2.48
2402	-2.09
553	-1.92

Table 3.8: Features with lowest (most negative) feature attribution on IOI in layer 6

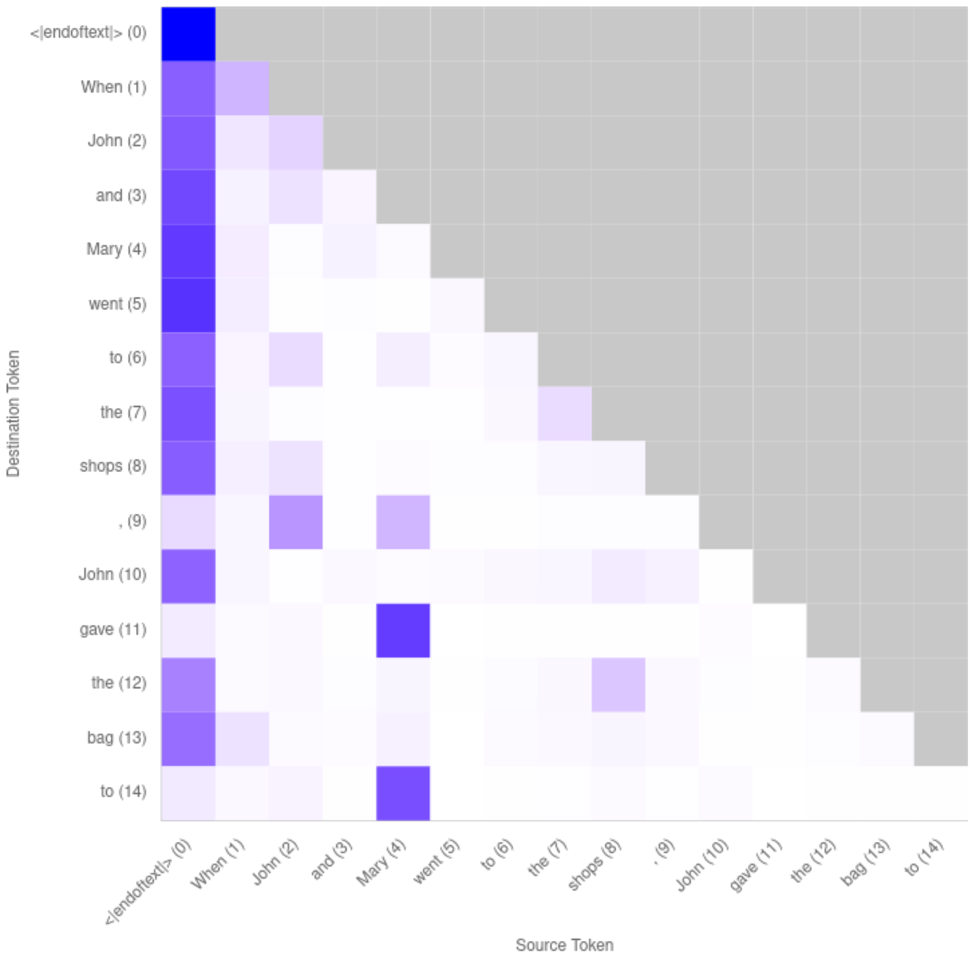

Figure 3.24: Attention pattern of H7.7 before ablating feature 1059 in layer 6.

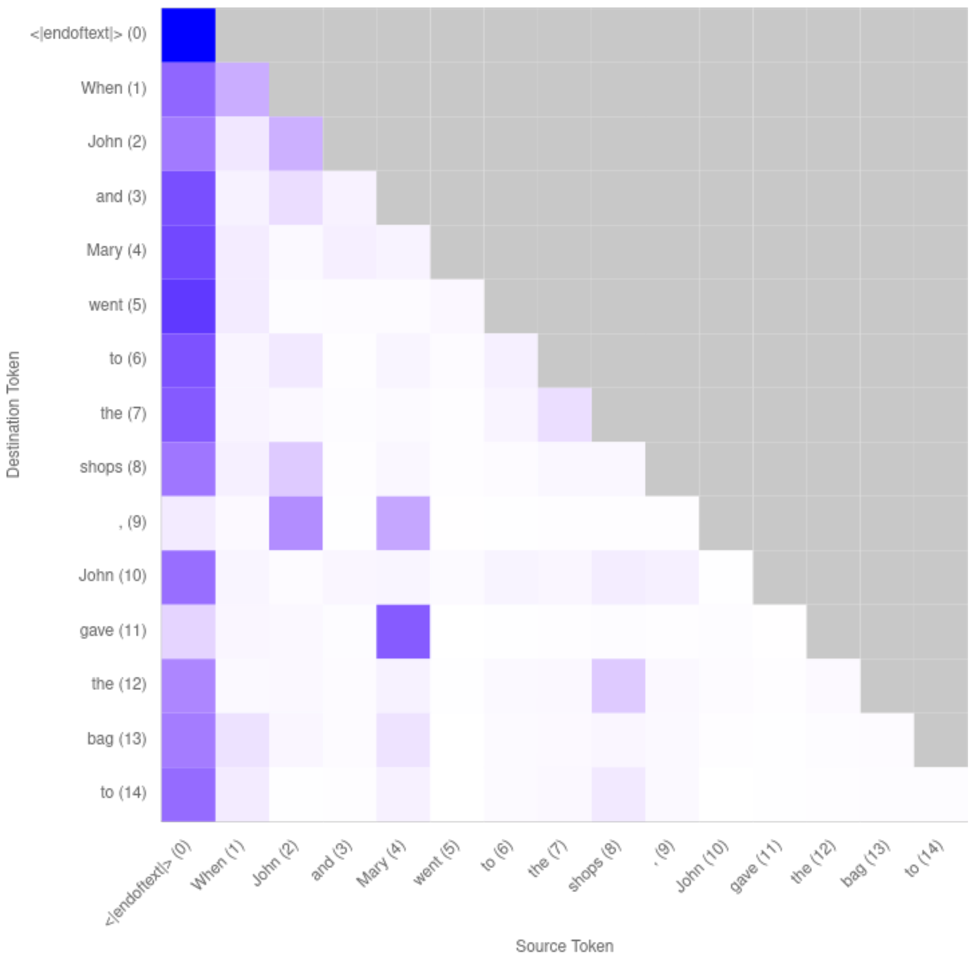

Figure 3.25: Attention pattern of H7.7 after ablating feature 1059 in layer 6.

Table 3.9: Max activating tokens with surrounding context for feature 2837 in layer 6. Token position is bolded.

token	context
Amy	, ready for another day of play and touch.<EOT> One day, a little girl named **Amy** went for a walk in
Lily	, there was a little girl named **Lily**. She loved to color
Mary	The end.<EOT> Once there was a girl named **Mary**. She was three years
Anna	everyone can be happy.<EOT> One day, a little girl named **Anna** went outside to play in
Lily	upon a time, there was a little girl named **Lily**. She was very curious
Molly	<EOT> Once upon a time, there lived a little girl named **Molly**. One day, Molly
Sarah	, brown brick in the park!<EOT> Once upon a time, there was a little girl named **Sarah**. She had to choose
Lily	.<EOT> Once upon a time, there was a little girl named **Lily**. She had a favorite
Lily	again.<EOT> Once upon a time, there was a little girl named **Lily**. She loved to play
Lily	him again.<EOT> Once upon a time, there was a little girl named **Lily**. She loved to ride
Lily	a new way.<EOT> Once upon a time, there was a little girl named **Lily**. She liked to play

3.5.3 Layer 7

The top activating features for layer 7 are listed in table 3.10. The only feature that had any significant feature attribution was feature 1854, which had an attribution score of 4.1, with the second highest being 0.07. The largest negative attribution score was 1053, with a score of -0.6. The top activating examples for 1854 are ones that activate at a token before "Mary" is predicted, indicating that this is a logit contribution feature. Multiplying the corresponding feature direction with the unembedding matrix I get that the two top logits are for "Mary" and " Mary", i.e. the same name without and without a space in front, after which are a bunch of other names, primarily girl names.

Feature	**Activation**
1854	0.90
1053	0.73
892	0.57
2678	0.37
2343	0.33

Table 3.10: Top activating features in layer 7 on IOI prompts.

The top activating examples for feature 1053 show that this feature activates on the "to" token in the context of someone giving something to someone else. This is interesting since this is the last layer and therefore there's no head that can use this feature for anything. Multiplying this feature direction with the unembedding matrix show that it doesn't contribute to any clear set of logits. One theory is that this feature is just left over from the previous layers, given that the network doesn't have infinite capacity to overwrite everything in the residual stream with new more important information.

Results

I'd say the main results of this thesis are:

- TinyStories-8M has a lot of standard attention heads, but importantly, it has no induction heads, which has been previously argued to be responsible for the majority of in-context learning [Olsson et al., 2022]. Still it is able to generate coherent and fluent text and perform in-context learning as shown in [Eldan and Li, 2023]. For example the IOI task without the use of such heads.
- TinyStories-8M has no negative name mover heads. Which provides insights into how the data distribution of the training data influences the emergence of certain classes of head: in this case heads that hedge the model predictions are less necessary given a more structured and predictable data distribution.
- TinyStories-8M implements a circuit for solving the IOI task which is very similar to the one found in GPT2-small by [Wang et al., 2022], although it is far cleaner and simpler, and doesn't involve any induction heads. Giving evidence for the universality of some transformer language model circuits.
- I find features in TinyStories-8M that are responsible for signalling to the name mover heads in the IOI circuit for toggling on and off the name moving behavior. This gives a better understanding of how the IOI circuit works in specific, but also attention heads in general: specific features written into token positions to trigger certain attention behaviors by specific heads.

Future Work

Ideas for what could be explored further:

- Explore the largest models in the tiny stories family to see if the same results hold. This would give evidence for the fact that the differences in the data distribution is what results in the differences from the IOI circuits of GPT2-small and TinyStories-8M.
- Investigate the features that trigger name mover heads to see specifically how they interact with the QK-circuit to toggle name attention.
- Train SAEs on earlier layers to see if we can equivalent features that trigger S-inhibition behavior.

Conclusion

This thesis has investigated the mechanisms underlying language model behavior, focusing on the Indirect Object Identification (IOI) task in TinyStories-8M and comparing it against GPT2-small. Through the application of established mechanistic interpretability techniques, we have gained valuable insights into how these models solve the IOI task and the similarities and differences in their internal circuits.

Our findings reveal that TinyStories-8M, despite being a much smaller model trained on simpler data, implements an IOI circuit remarkably similar to that of GPT2-small. However, TinyStories-8M's circuit is notably cleaner and simpler, lacking the negative name mover heads found in GPT2-small. This difference suggests that heads that hedge model predictions may be less necessary given a more structured and predictable data distribution.

A significant finding is that TinyStories-8M achieves in-context learning without relying on induction heads, challenging existing assumptions about the necessity of certain architectural features for this capability. Through sparse autoencoders, we identified specific features responsible for signaling name-moving behavior to attention heads within the IOI circuit, providing deeper insights into attention mechanisms.

The similarities in the IOI circuits between TinyStories-8M and GPT2-small, despite their differences, provide evidence for the universality of certain transformer language model circuits. This suggests that some fundamental computational patterns may emerge across different scales and training regimes.

Our research contributes to the field of mechanistic interpretability by offering a comparative analysis of circuit formation in models of different scales and training regimes. Future work could explore larger models in the TinyStories family and investigate features that trigger specific attention behaviors.

In conclusion, this thesis advances our understanding of the internal mechanisms of language models, challenges some existing assumptions, and provides a foundation for future work in understanding the use of features as computational units in transformer circuits.

Bibliography

Sid Black, Lee Sharkey, Leo Grinsztajn, Eric Winsor, Dan Braun, Jacob Merizian, Kip Parker, Carlos Ramón Guevara, Beren Millidge, Gabriel Alfour, et al. Interpreting neural networks through the polytope lens. *arXiv preprint arXiv:2211.12312*, 2022.

Trenton Bricken, Adly Templeton, Joshua Batson, Brian Chen, Adam Jermyn, Tom Conerly, Nick Turner, Cem Anil, Carson Denison, Amanda Askell, et al. Towards monosemanticity: Decomposing language models with dictionary learning. *Transformer Circuits Thread*, 2, 2023.

Lawrence Chan, Adrià Garriga-Alonso, Nicholas Goldwosky-Dill, Ryan Greenblatt, Jenny Nitishinskaya, Ansh Radhakrishnan, Buck Shlegeris, and Nate Thomas. Causal scrubbing, a method for rigorously testing interpretability hypotheses. *AI Alignment Forum*, 2022. `https://www.alignmentforum.org/posts/JvZhhzycHu2Yd57RN/causal-scrubbing-a-method-for-rigorously-testing`.

Arthur Conmy, Augustine Mavor-Parker, Aengus Lynch, Stefan Heimersheim, and Adrià Garriga-Alonso. Towards automated circuit discovery for mechanistic interpretability. *Advances in Neural Information Processing Systems*, 36:16318–16352, 2023.

Hoagy Cunningham, Aidan Ewart, Logan Riggs, Robert Huben, and Lee Sharkey. Sparse autoencoders find highly interpretable features in language models. *arXiv preprint arXiv:2309.08600*, 2023.

Ronen Eldan and Yuanzhi Li. Tinystories: How small can language models be and still speak coherent english? *arXiv preprint arXiv:2305.07759*, 2023.

Nelson Elhage, Neel Nanda, Catherine Olsson, Tom Henighan, Nicholas Joseph, Ben Mann, Amanda Askell, Yuntao Bai, Anna Chen, Tom Conerly, Nova DasSarma, Dawn Drain, Deep Ganguli, Zac Hatfield-Dodds, Danny Hernandez, Andy Jones, Jackson Kernion, Liane Lovitt, Kamal Ndousse, Dario Amodei, Tom Brown, Jack Clark, Jared Kaplan, Sam McCandlish, and Chris Olah. A mathematical framework for transformer circuits. *Transformer Circuits Thread*, 2021. https://transformer-circuits.pub/2021/framework/index.html.

Nelson Elhage, Tristan Hume, Catherine Olsson, Nicholas Schiefer, Tom Henighan, Shauna Kravec, Zac Hatfield-Dodds, Robert Lasenby, Dawn Drain, Carol Chen, et al. Toy models of superposition. *arXiv preprint arXiv:2209.10652*, 2022.

Nelson Elhage, Robert Lasenby, and Christopher Olah. Privileged bases in the transformer residual stream. *Transformer Circuits Thread*, page 24, 2023.

Joshua Engels, Isaac Liao, Eric J Michaud, Wes Gurnee, and Max Tegmark. Not all language model features are linear. *arXiv preprint arXiv:2405.14860*, 2024.

Tom Lieberum, Matthew Rahtz, János Kramár, Neel Nanda, Geoffrey Irving, Rohin Shah, and Vladimir Mikulik. Does circuit analysis interpretability scale? evidence from multiple choice capabilities in chinchilla. *arXiv preprint arXiv:2307.09458*, 2023.

Kevin Meng, David Bau, Alex Andonian, and Yonatan Belinkov. Locating and editing factual associations in GPT. *Advances in Neural Information Processing Systems*, 36, 2022. arXiv:2202.05262.

Neel Nanda, Lawrence Chan, Tom Lieberum, Jess Smith, and Jacob Steinhardt. Progress measures for grokking via mechanistic interpretability. *arXiv preprint arXiv:2301.05217*, 2023.

nostalgebraist. Interpreting GPT: the logit lens. `https://www.lesswrong.com/posts/AcKRB8wDpdaN6v6ru/interpreting-gpt-the-logit-lens`, 2020. Accessed: [2024].

Chris Olah, Arvind Satyanarayan, Ian Johnson, Shan Carter, Ludwig Schubert, Katherine Ye, and Alexander Mordvintsev. The building blocks of interpretability. *Distill*, 2018. doi: 10.23915/distill.00010. https://distill.pub/2018/building-blocks.

Chris Olah, Nick Cammarata, Ludwig Schubert, Gabriel Goh, Michael Petrov, and Shan Carter. Zoom in: An introduction to circuits. *Distill*, 5(3):e00024–001, 2020.

Catherine Olsson, Nelson Elhage, Neel Nanda, Nicholas Joseph, Nova DasSarma, Tom Henighan, Ben Mann, Amanda Askell, Yuntao Bai, Anna Chen, et al. In-context learning and induction heads. *arXiv preprint arXiv:2209.11895*, 2022.

Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. 2018.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

Senthooran Rajamanoharan, Arthur Conmy, Lewis Smith, Tom Lieberum, Vikrant Varma, János Kramár, Rohin Shah, and Neel Nanda. Improving dictionary learning with gated sparse autoencoders. *arXiv preprint arXiv:2404.16014*, 2024.

Aaquib Syed, Can Rager, and Arthur Conmy. Attribution patching outperforms automated circuit discovery. *arXiv preprint arXiv:2310.10348*, 2023.

Adly Templeton. *Scaling monosemanticity: Extracting interpretable features from claude 3 sonnet.* Anthropic, 2024.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Kevin Wang, Alexandre Variengien, Arthur Conmy, Buck Shlegeris, and Jacob Steinhardt. Interpretability in the wild: a circuit for indirect object identification in gpt-2 small. *arXiv preprint arXiv:2211.00593*, 2022.