Continuing Commmon Sequences

Link to the Colab Notebook:
https://colab.research.google.com/drive/1Qib9r1AuRDecYy882MkZe461-mjX_YSL?usp=sharing

This is one of the 200 Concrete Open Problems in Mechanistic Interpretability highlighted by Neel Nanda in one of his posts. The problem is about continuing common sequences in natural language by the language model, but the sequence should not be present prior in the prompts. Some common examples of sequences present in natural language are as follows:

- "1, 2, 3, 4, " → completed by '5'
- "January, February, March, " → completed by 'April'
- "Alpha, beta, " → completed by 'gamma'

Now, the work I have done on this task is mostly centered around Edge Attribution Patching and the corresponding computational graphs and scores related to GPT2-small. The second half of the notebook is similar what Neel has done in case of IOI, I am doing activation patching and checking the attention head pattern for the sequences. There is also the visualizations in case of activation and attribution patching for the above task.
[I wanted to do more experimentation with mlp patching, activation patching and comparisons between activation and attribution patching for the above task, but due to time constraints at the moment I couldn't do so. (Because I have my mid-semester exams going on right now, in my college). But I will surely come back to this problem and try to experiment more on it.]
Here is how i have decided to tackle the problem of Continuing Common Sequences (in the same order as in colab notebook):

(1) <u>The Dataset:</u>

For EAP tasks I took the dataset in this form:

```
clean        1,2,3,4,
corrupted    1,1,1,1,
target               5
Name: 0, dtype: object
```

Here 'clean' is the real sequence we want GPT2-small to complete and target is the real answer for completing it. I will use this target token to also compute performance of the model for the above task pre and post edge-ablation. Also, I chose the corrupted sequence in a way so that it always repeats the initial element in the sequence, the length of this sequence will obviously be the same as the clean sequence. Now, the question is why I chose to use corrupted sequence in this specific way and the answer is

it seemed the right way. Since both ‘clean’ and ‘corrupted’ are sequences of their own and the special thing about ‘corrupted’ is that it is always a constant sequence.

(2) Baseline Scoring:

There are usually two methods for calculating the baseline score against which the model will be evaluated when we do edge attribution patching: One is evaluate_baseline score and the other one is evaluate_graph score. Now in case of evaluate baseline, for each example, I am calculating the sum of probabilities of predicted tokens which are equal to the target token (the actual next element in the series) and then subtracting the sum of probabilities of wrongly predicted tokens. This needs to be done for every example in the dataset and then the probability difference is averaged over the entire dataset and that particular score is our evaluate_baseline score. The number of top predictions I am considering here is equal to 10 because the sum of those top 10 predicted tokens will amass to nearly 90 percent of the whole. This evaluate_baseline function is used to basically get the performance of the actual model (GPT2-small) for doing the certain task, which is continuing common sequences in our case.
Now in case of evaluate_graph function the things are done differently. The metric is same as discussed above but here the corrupted series comes into play. We have the argument of graph (g) that we can pass to this function and also if we decide to ablate some edges in the graph we can easily do that and pass the ablated graph of the model and again calculate the score. To get the baseline scoring for eap, we pass the unaltered graph to the model and get the evaluate_graph score.

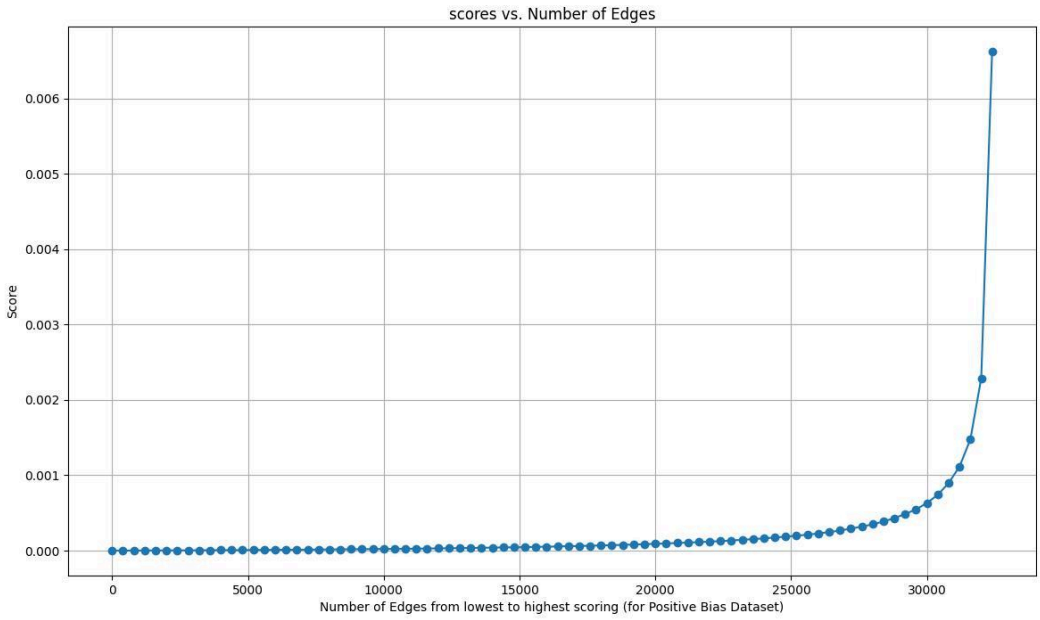
(3) Graph evaluation:

The evaluate graph function implementation underlying logic is as follows: we have an edge mask tensor of shape (# of sending nodes, # of receiving nodes), where each entry indicates whether the edge is in the circuit. We then create a tensor of component activation differences of dim (# of sending nodes, model dim). This will hold the corrupted activations minus the clean activations of every sending node. We first run the model on the corrupted examples, collecting the activations in the tensor. We then run the model on the clean input. During the forward pass, we subtract each sending node's output from its corrupted output in the storage tensor, to construct this tensor of differences. At the same time, we reconstruct the input to each node v according to the edge mask: if an edge (u,v) is not in the graph, we add in the activation difference of u, effectively removing the clean activation and adding the corrupted activation; the edge is thus corrupted.

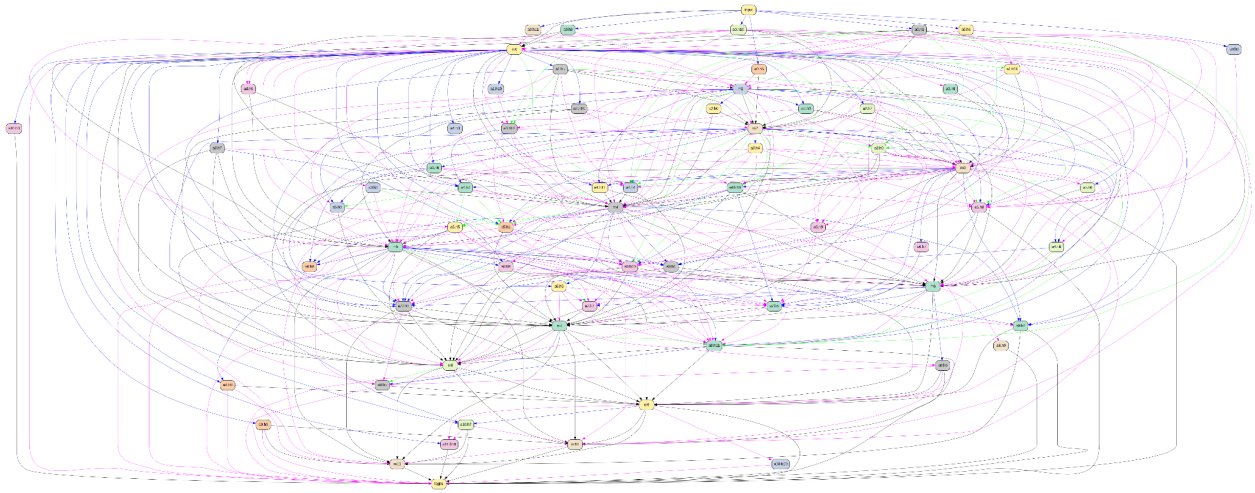
(4) Edge attribution Patching:

We start by running the attribute function which starts calculating the edge scores for every edge in the graph. It calculates this scores based on an approximation of the difference between the losses when we replace the activations of the edge during the clean run with the activations of the same edge (collected during the corrupted run). Now since we have the edge scores of every edge in the model, we can do ablation and all based upon what we want. In case of GPT2-small we have 158 nodes and 32491 edges

and after scoring each of these edges using eap , we have the edge-score distribution as follows:



In the above figure we see a plot of edge score (collected from eap) versus the number of edges in the computational graph of the model. Note that I have arranged the edge scores and the edges in the ascending order here (gaps between points is equal to 400 edges) and one interesting thing we can conclude here is that we can ablate nearly 30000 edges whose score is less than 0.001 and the model will still do good in the above mentioned task.
Here is the circuit for GPT2-small if we decide to keep only the 500 best scoring edges in the graph. This circuit is formed with the help of (graph)apply_threshold function where we are considering all the edges whose score is greater or equal to the 500 greatest absolute score. This circuit has 428 edges in total.

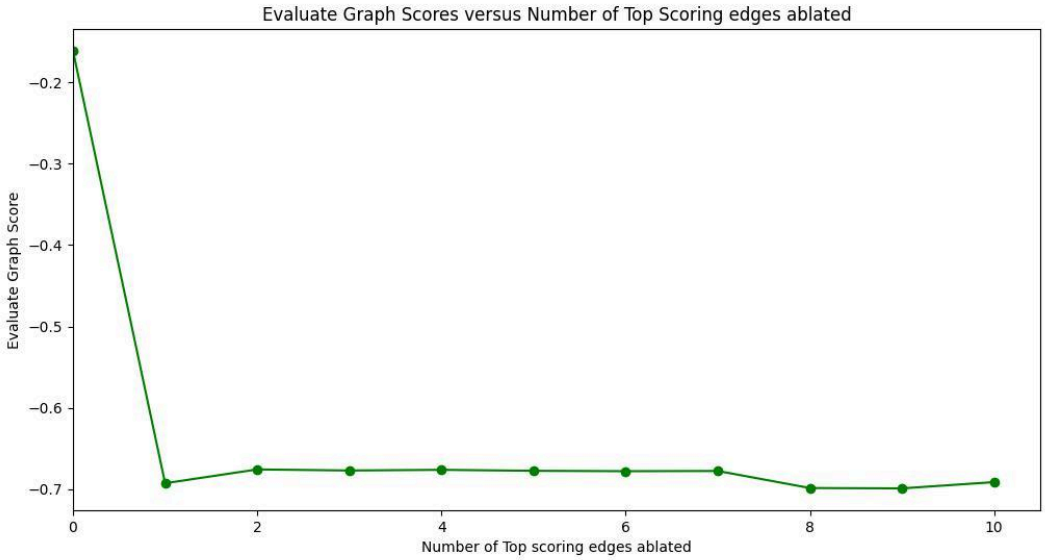


Now to prove my point that how less important the least scoring edges are, I have ablated all the edges except top 500 best scoring edges and the evaluate graph score for ablated circuit is: `-0.7030360698699951` and if we ablate all the edges except the top 1500 best scoring edges, the evaluate graph score for this one is `-0.7418543696403503`, which looks pretty interesting to me as the difference between these two scores is < 0.1.

Ablating the top scoring edges (Important Observations):

Now, here I have tried to do a different thing by manually ablating the top scoring edges, to check the robustness of our model (gpt2-small). I observed that as we manually ablate more and more top scoring edges the evaluate graph score gets more and more negative. This is because the metric calculates summation of correct probabilities minus incorrect probabilities and as we ablate the top scoring edges the circuit loses its ability to predict correct output for the sequence and hence probabilities of incorrect next-predicted-tokens increases while as probabilities of correct next-predicted-tokens decreseas. In this case I have only checked for the Top 10 edges of GPT2-small, and the observation is quite interesting. If we only ablate the top 1 edge the score drops significantly but if we drop top 2 or top 3 or top n (n=10) edges collectively, the scores drop is roughly equivalent to dropping the first edge only, which shows the significance of the top scoring edge (input->m0) in the circuit. We can see this in the below plot of evaluate_graph scores versus the number of top scoring edges ablated. There is a sharp decrease in the score after ablating first edge and then the score remains almost constant. The reason for this kind of observation (importance of input->m0) is the same as discussed by Neel in his IOI tutorial. Here it is “The main place attribution patching fails is MLP0 and the residual stream. Attribution patching works badly for "big" things which are poorly modelled as linear approximations, and works well for "small" things which are more like incremental changes. Anything involving replacing the embedding is a "big" thing, which includes residual streams, and in GPT-2 small MLP0 seems to be

used as an "extended embedding" (where later layers use MLP0's output instead of the token embedding), so I also count it as big." Hence from this discussion we infer how important MLP0 (m0) is for the successive layers and if we ablate the 'input->m0' the model performance is badly hurt.



Here are the Top 3 best scoring edges in the computational graph of GPT2-small for the task of continuing the common sequences:

```
Edge(input->m0)       Score: 0.08633261173963547
Edge(m11->logits)     Score: 0.04759420081973076
Edge(m0->a4.h10<q>)   Score: 0.04257625341415405
```

The edges are highlighted with their corresponding eap scores.

- ## Attention & Attribution Patching (Analogous to Neel's IOI task):

  (a) Dataset:

  The dataset for the following tasks is a bit different from the what I have been using in the tasks discussed above. We are also including a corrupted target here which is just an element of the corrupted series (constant series). This will be used to compare the probability difference when I patch activations from the clean run into the corrupted run and analyze how the output has flipped (in terms of probability differnce) as i patched those activations. The dataset here is a bit smaller because we need the target and corrupted_target to be of single token length.

	clean	corrupted	target	corr_target
0	1,2,3,4,	1,1,1,1,	5	1
1	1 2 3 4	1 1 1 1	5	1
2	I,II,III,IV,	I,I,I,I,	V	I

(b) <u>The metric:</u>

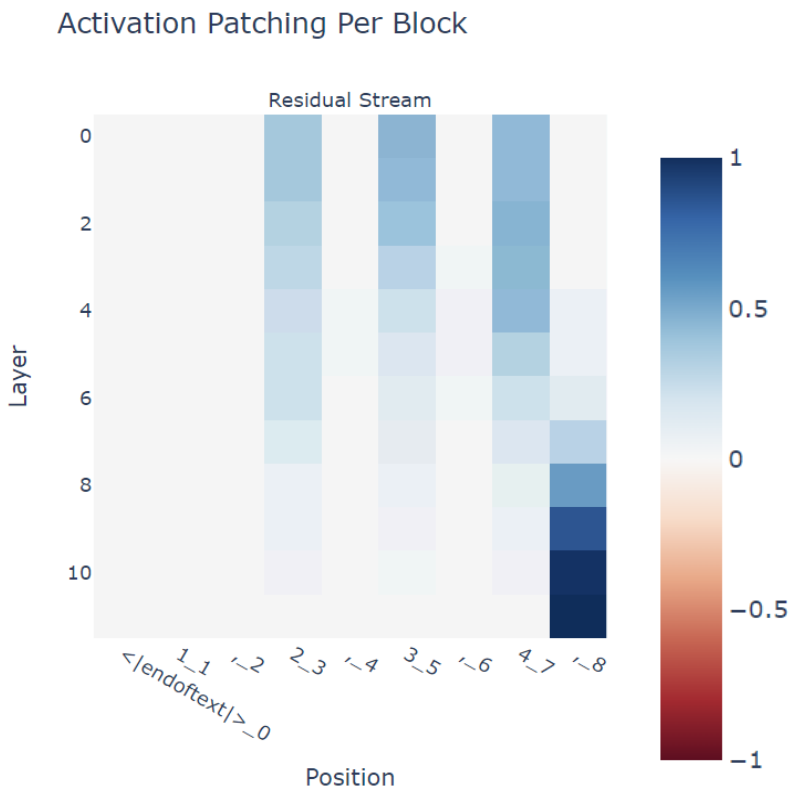
Here aslo I am using probability difference, which is equal to the difference in probabilities between the probabilities of correct and incorrect predictions of a series according the the values set in 'target' and 'corr_target' columns of the dataset. For example, if we calculate this metric for clean series: "1, 2, 3, 4, " the correct probabilities will be the probabilities of predictions equal to target = 5 and the incorrect probabilities will be the probabilities of predictions equal to corr_target = 1 and hence probability difference will be the difference between these two values. Now, the actual metric we are using for analysis is a probability difference normalized between -1 and +1. This thing we do by setting a Clean Baseline and a Corrupted Baseline, where Clean Baseline is the probability difference run on all the clean examples in the dataset and the Corrupted Baseline is the probability difference run on all the corrupted exammples in the dataset.

Hence for a new example, we would calculate the metric as follows:

METRIC = [PROB. DFFERENCE (new example) - CORRUPTED_BASELINE] / [CLEAN_BASELINE - CORRUPTED_BASELINE]

(c) <u>Activation Patching:</u>

In Activation Patching, we run the mode on the clean prompts and cache the clean activations. Now, we run the model on the corrupted prompt and start patching the clean activations in this corrupted run at different layers and different positions. If the prompts are similar upto a certain position, nothing changes, but after the clean and corrupted prompts start to vary, activation patching starts to kick in. So we do this by residual stream patching i.e., for a specific layer L and position P, we run the model on the corrupted prompt. Till layer L it's unchanged, then at layer L we patch the clean residual stream in at position P and replace the original residual stream. The run then continues as normal, and we look at the patch metric (probability difference). We iterate over all layers L and positions P.

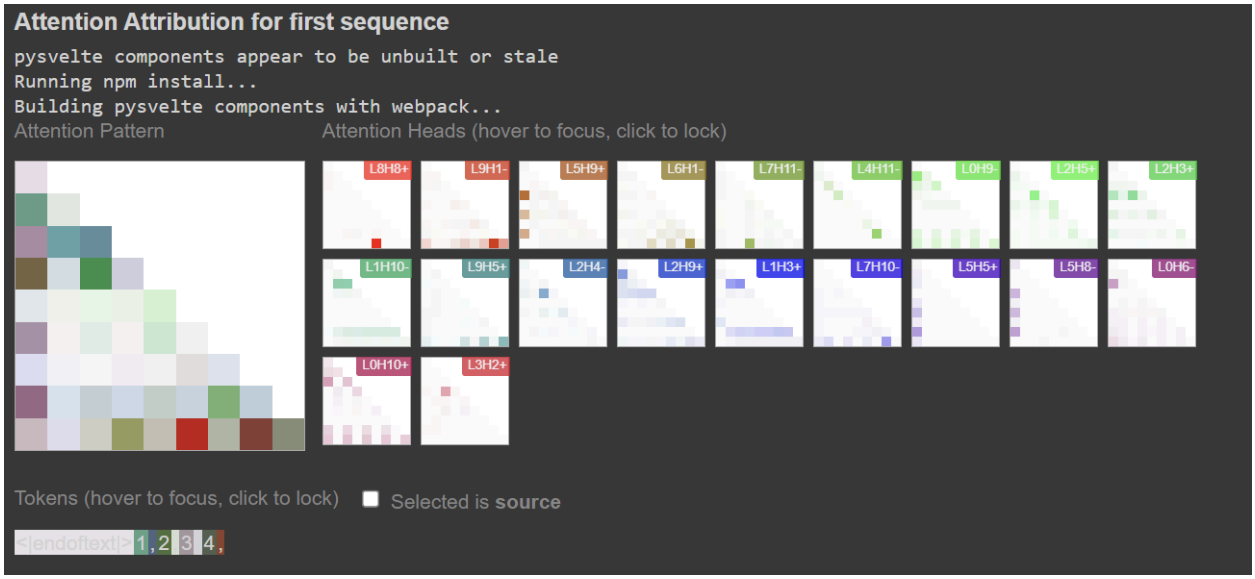


Looking at the Activation Patching pattern above, we realize that, since clean prompt “1, 2, 3, 4, ” and corrupted prompt “1, 1, 1, 1, ” differ from the second element of the series itself, therefore things are stored at those positions (elements 2,3,4) from the early layers till layers 5,6. Then an immediate transition is happening at the end token and things are shifted to the later attention layers (layers 6 to 11). The interesting thing about it is that there is an immediate shift of information between layers 0-6 and layers 7-11 and the information does not flow through any intermediary tokens.

(d) <u>Attention Attribution:</u>

The first thing we should try is to look at the linear approximation of what happens if we set each element of the attention pattern to zero. The important thing to note is that this is very different from what we are going to do later with activation patching. In activation patching we will be setting up a counterfactual using clean and corrupted prompts while in this case we are just looking at what is going on in this problem. We are not controlling anything here unlike activation patching whee we patch clean activations onto a corrupted run and try to figure out whether that flips the output towards the clean-output. Since we are probability difference as our metric, and each example in the dataset is independent, so we can run this attention attribution on a single prompt from the dataset, as well as the entire dataset and taking the average of the probability difference.

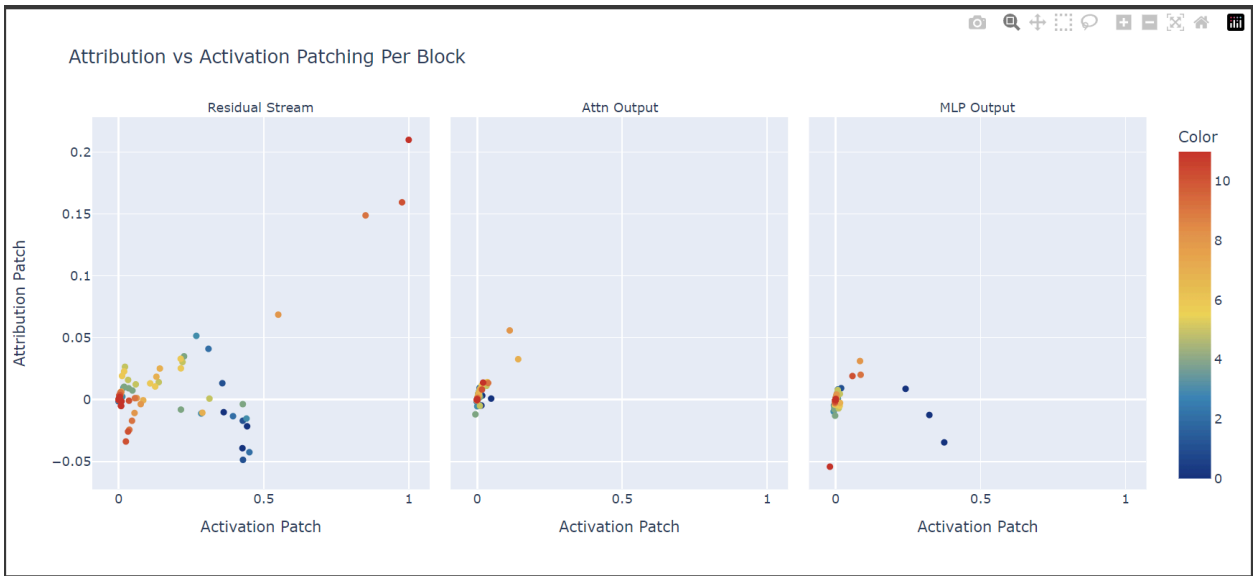
We plot the attention attribution patterns and scale them to be in [-1, 1], split each head into a positive and negative part (so all of it is in [0, 1]), and then plot the top 20 head-halves by the max value of the attribution pattern. Here is the visulaization of the attention pattern of GPT2-small for the above mentioned task of completing the sequences but only for the first example i.e., completing the sequence “1, 2, 3, 4, ” → ‘5’. (Better look them up in the notebook for interactive plots)



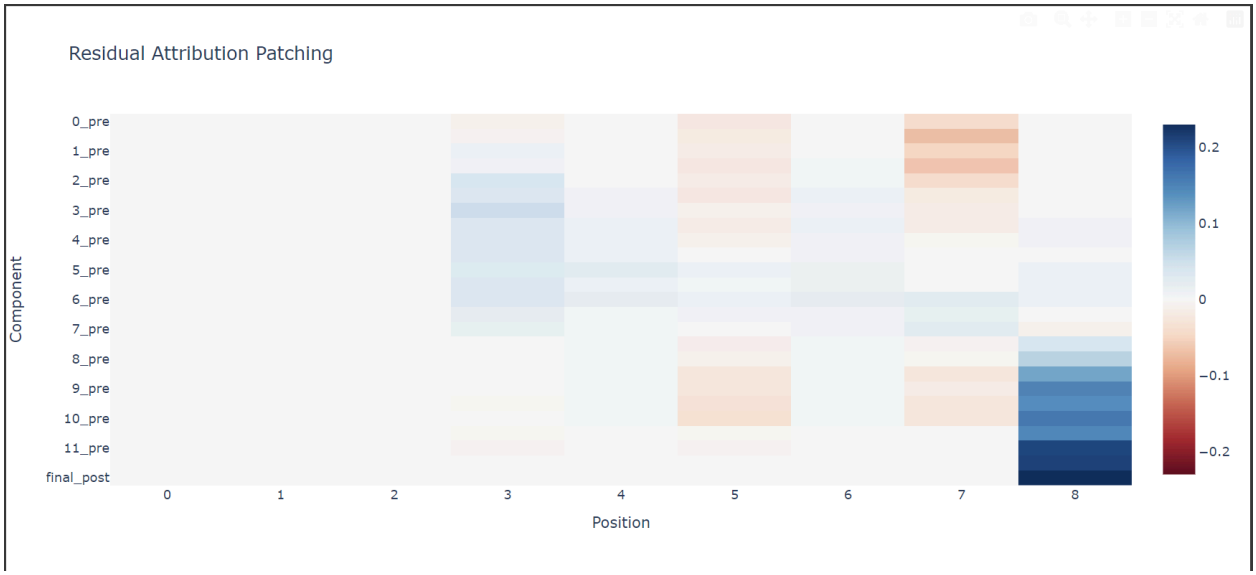
Now here is the visualization of attention pattern for GPT2-small for the whole dataset of common sequences.

- **<u>Comparing activation and attribution patching:</u>**
  The critical insight here is that attribution patching serves as a poor approximation for large-scale activations, such as residual stream patching and MLP0 (m0), while offering a reasonable approximation for other layer outputs, attention head outputs, and head activations (including queries, keys, values, and attention patterns). This approach helps us recover the general structure of the IOI circuit. The scatter plot of activation versus

attribution across blocks, residual stream outputs, attention layer outputs, and MLP layer outputs reveals a significant degradation in performance for residual streams and MLP0, with more accurate results observed for other components.



The primary issue stems from the fact that attribution patching is a linear approximation, which tends to perform better for smaller perturbations. The role of LayerNorm complicates this further, as it normalizes activations based on their relative scale within the residual stream. Consequently, patching the entire stream or early layer outputs produces poor results.



Moreover, LayerNorm introduces additional complexity into the residual stream. In the case of GPT-2 Small, as previously discussed in "ablating the top scoring edges section,

MLP0 (m0) plays a distinct role as an extension of the embedding space, effectively breaking the symmetry between the tied embedding and unembedding layers. This leads later layers to rely on MLP0 as a substitute for token embeddings.