Embedding and Transformer Synthesis

Rick Goldstein

7/14/23 - 7/16/23
Alignment Jam 3.0

Quick Summary:

Prior to this hackathon, I had a (low-level) mechanistic interpretability hypothesis about how a specific 1-layer transformer trained on a specific classification function might work.

For this hackathon:

- I programmatically created a set of embeddings that can be used to perfectly reconstruct the original classification function ("embedding synthesis").
- I used these embeddings to programmatically set weights for a 1-layer transformer that can perfectly reconstruct the original classification function ("transformer synthesis"). With one change, this reconstruction matches my original hypothesis of how the pre-existing transformer works.
- I ran several experiments on my synthesized transformer to begin to evaluate my hypothesis. I am slighty less confident in my original hypothesis as a result of the experiments (though I think it may be close).
- A Jupyter notebook can be found at https://github.com/freestylerick/alignmentjam3.

Background & Contribution:

It is difficult to interpret the weights inside of deep learning models and as a result, we cannot ensure that models will not partake in harmful behaviors. However, if we can programmatically set the weights inside of a deep learning model (even if starting with small components of a model), we will have a better sense of the logic occuring inside of the model and have a better chance of preventing these undesired behaviors.

Several past examples of programmatically setting weights programmatically come to mind:

- In 2020, OpenAI demonstrated that InceptionV1 had many curve detectors; Chris Olah manually created multiple families of curve detectors.
- Introduced more recently, RASP is a symbolic programming language that allows us to write some computational steps using transformer primitives. tracr compiles these RASP programs into actual transformers. Dan Friedman et al converts RASP programs to human-readable code.
- Two recent papers interpret transformer network(s) trained on modulo addition (Neel Nanda et al, Ziqian Zhong/Ziming Liu et al).

Prior to this hackathon, I spent ~2.5 weeks exploring Stephen Casper's Transformer mechanistic interpretability challenge (#2) (I'll explain this more in the next section). While this problem has been solved* by Stefan Heimersheim and Marius Hobbhahn, there are several unexplained low-level questions such why model errors occur around the decision boundaries. Additionally, I think there is a lower-level interpretation that has not fully been explored (for example, I have detected patterns in attention (QK) circuits that are deserving of a separate discussion).

My consideration of the above unanswered question (why model errors occur around boundaries) and an Anthropic-style weight exploration of low-level circuits have led to a hypothesis about how the model weights work together to achieve the behaviors we are seeing.
For this hackathon, I wanted to examine

1) If my hypothesis of how the trained transformer model works is actually a complete description that can implement the original classification function.
2) If I could actually code up these weights into a 1-layer transformer network.
3) If my transformer seems to share properties of the original transformer trained for Stephen's challenge.
4) If I can explore why model errors occur around the decision boundaries.

To address these:

1) I computed embeddings that can perfectly implement the original classification function (but needed to add one unanticipated step, explained later).
2) I built a fully functioning transformer that perfected implements the original classification function. The transformer aligns with my hypothesis of what the pre-existing transformer is doing (with this one change).
3) I ran linear probe experiments to show that my transformer relies more on the MLP portion than the original transformer; this slightly weakens my confidence in my hypothesis.
4) I ran an experiment to examine model errors around decision boundaries.

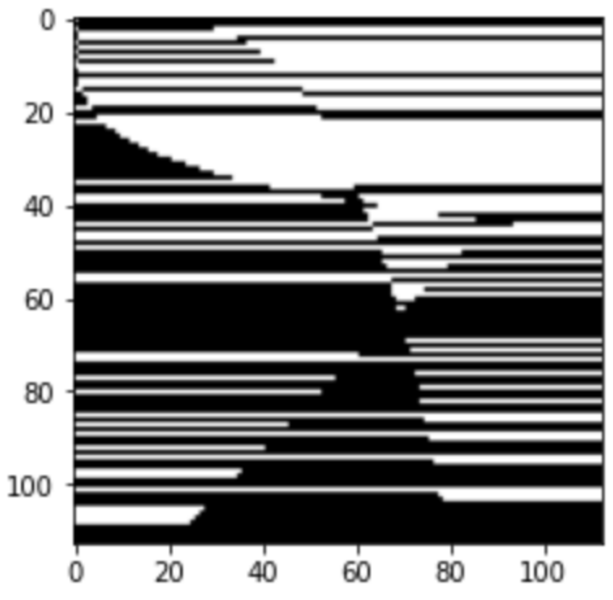
In my mind, the first two contributions which together programmatically create a usable and interpretable transformer are significantly more important than the other two contributions.

My code has been uploaded at https://github.com/freestylerick/alignmentjam3. Work was conducted on a Dell Windows XPS17 laptop with an NVIDIA GeForce RTX4080 with 12 GB GDDR6 GPU. Bugs, errors, and omissions are quite likely given the limited timeframe.

Original Transformer & My Hypothesis:

Here is a 113 x 113 binary labelling function. The code (from a Stephen Casper post) is presented on the left and a visualization of the function is on the right (also from a Stephen Casper post):

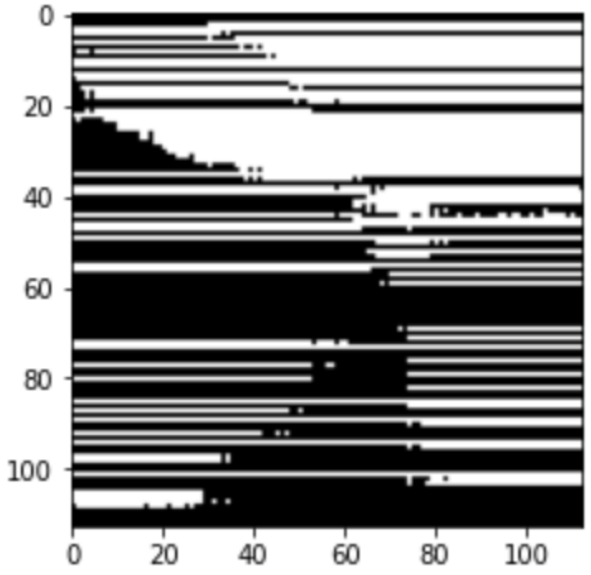
```
p = 113
def label_fn(x, y):
  z = 0
  if x ** 2 < np.sqrt(y) * 200:
    z += 3
  if y ** 2 < np.sqrt(x) * 600:
    z -= 2
  z += int(x ** 1.5) % 8 - 5
  if y < (p - x + 20) and z < -3:
    z += 5.5
  if z < 0:
    return 0
  else:
    return 1
```



There are four high level ideas I'd like you to take away from the function:

1) The modulo ("% 8") on line 08 makes the function periodic, having different patterns of black and white lines across different rows of the above plot. Some rows are all black. Some all white, some black then white, some white then black, some white then black then white, and some black then white then black.
2) There are three interesting inequalities of x and y (lines 04, 06, 09). Switches from black to white or white to black occur around these functions. I will refer back to these as **the three interesting inequalities**.
3) More logistical - X is along the vertical axis. For example, the bottom row that is all black is saying that when x=112, label_fn(x, y) is 0 for all values of y.
4) The function only takes discrete inputs: x,y E [0,112].

These components will feed into both my hypothesis as well as my synthesis of embeddings.



For the challenge, Stephen trained a network consisting of an embedding layer, a 1-layer 8-head transformer layer, an MLP layer, and an unembedding layer. This model was trained on about half of the data points. The network's predictions can be viewed on the left. The model begins by embedding x, y, and a third dummy term (z). After applying the transformer and MLP layers, output is read off by unembedding z.

I will now present my hypothesis about how the network works. To be clear, the formulation of this hypothesis occurred prior to the hackathon and is deserving of a separate post (this is **extremely** messy and incomplete but it presents a few inspirations for this hypothesis). My intention for this hackathon is not to explain how I came to the hypothesis, but rather to provide enough detail that you understand the hypothesis and how it aligns with my embedding synthesis below. It is reasonable for you to be skeptical of this hypothesis given my limited support.

I hypothesize that the network is trying to learn how to count in the y-dimension while each row (each x-dimension) learns specific y-cutoffs based where it should switch between black and white based on the three interesting inequalities. By matching the y-cutoffs (from the x-dimension) with the value of y (from the y-dimension), the model is able to learn the high level structure of the labelling function and to mostly learn the curvature of the three interesting inequalities.

To make this slightly more concrete, the row x = 40 should be black from 0 to 64 and white from 65 to 112. The network learns an embedding for x = 40 of 64.5. Then combined with y's unchanged value, $E_y(y) - E_x(x)$ will be negative from [0, 64] and positive from [65, 112]. (Note, I am assuming that $E_y(y) = y$). It is more complicated for rows that have two switches (e.g. black to white back to black). One more note - if it seems like I'm relying a lot on embeddings, that's intended. The proposed solution to the challenge argues (among other things) that most of the importance comes from “extended embeddings” of x and y. The above authors use the term “extended embeddings” refer to the residual stream of z after applying an fixed attention pattern (resid_mid).

A few more quick thoughts - the transformer layer moves info about x and y from their original locations into the residual stream for z. However, I believe that the model fails to learn how to exactly count in the y-dimension, or in other words $E_y(y) \approx ay$ but $E_y(y) \neq ay$. This failure is causing model errors near the three interesting inequalities (where a is a proportionality term).

To wrap up this section, I want to emphasize that this hypothesis lacks support (so it’s completely reasonable if you are skeptical) and is probably at least partially wrong. The goals of this hackathon are not to generate a good hypothesis, but rather to take an existing hypothesis and synthesize embeddings and a transformer deep network based on a hypothesis.

Embedding Synthesis:

I now describe how I create embeddings. The goal of the embeddings will be to do a similar process as described in my hypothesis above. For each row X, I will learn multiple cutoff points. When a specific cutoff point is combined with each Y, the output will be positive when the labelling function is 1 and the output will be negative when the labelling function is 0. The ultimate goal of this section we are working towards is as follows: **we want to be able to create embeddings such that when combined, the embeddings perfectly recreate label_fn(x, y).**

I now present a deeper dive into how I calculate a single embedding value and why it works.

Earlier I noted that for X = 40, we select the embedding of E(X) = 64.5.
At (x = 40, y = **64**), x ** 2 < np.sqrt(y) * 200 is false.
At (x = 40, y = **65**), x ** 2 < np.sqrt(y) * 200 is true.
64.5 is the midpoint where this function switches parity.

Also note that the overall labeling function also switches it’s overall parity between these two points (in this case, from false to true). Because of the modulo function, there is not a guarantee that the network will switch parity whenever one of the interesting inequalities switches parity. We must also check that the network switches it’s overall parity between these points as well. To be clear - I care about the network switching parity and if it’s from true to false or false to true. I also care that the inequality switches, but don’t care about which direction it switches.

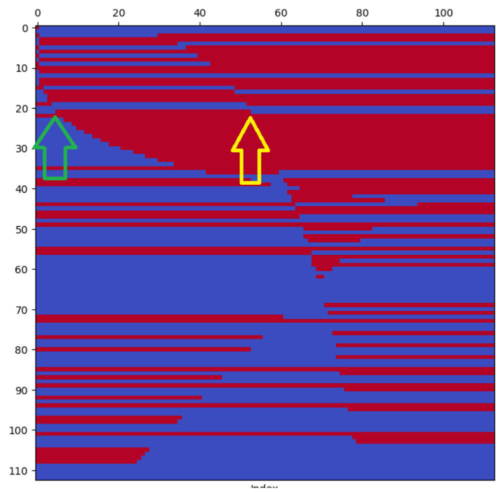
Consider the below table for X = 40 and different values of Y. Note how the below set of embeddings successfully matches the model with the true label for all values. Lines where the model switches from False to True are highlighted in yellow.

X	Ex(X)	Y	Ey(Y)	Ey(Y)-Ex(X)	model label	true label	match?
40	64.5	0	0	-64.5	FALSE	FALSE	TRUE
40	64.5	1	1	-63.5	FALSE	FALSE	TRUE
…	…	…	…	…	…	…	…
40	64.5	62	62	-2.5	FALSE	FALSE	TRUE
40	64.5	63	63	-1.5	FALSE	FALSE	TRUE
40	64.5	64	64	-0.5	FALSE	FALSE	TRUE
40	64.5	65	65	0.5	TRUE	TRUE	TRUE
40	64.5	66	66	1.5	TRUE	TRUE	TRUE
40	64.5	67	67	2.5	TRUE	TRUE	TRUE
…	…	…	…	…	…	…	…
40	64.5	111	111	46.5	TRUE	TRUE	TRUE
40	64.5	112	112	47.5	TRUE	TRUE	TRUE

Ultimately, these X embeddings are specific to each function and there will be one value for each integer (X) value from 0 - 113. However, each function actually needs 8 such embeddings: There are 3 binary rules we must consider when calculating such embeddings giving us 8 different cases. (As an aside - when I started coding on Friday, I overlooked one of these rules, and thus my original hypothesis that I set out to code did not lead to an embedding that correctly classified all points).

First, we must consider for each function's whether we are moving from an overall 1 label to an overall 0 (I refer to this situation as *false*) *or* vice versa (I refer to this as *true*).
Second, we want a separate embedding for the *left* of the function and the *right* of the function. This will allow us to use the network's MLP layer.
Third, for a given direction (e.g. *left*), we must consider there is another function that changes parity before getting to the boundary of Y's input domain ([0, 112]). When moving to the border, I call this *wall*. When moving to another function, I call this *center*.

Here is an example of *wall* vs *center*: Consider the row X = 21 which as we move from left to right, changes parity from False (blue) to True (red) at the green arrow. Then it goes back to False at the yellow arrow. At the green arrow, we want to one embedding for the *false**left*wall* embedding for the inequality x ** 2 < np.sqrt(y) * 200 and another embedding for *true*right*center* again for x ** 2 < np.sqrt(y) * 200. (Moving right takes us to the yellow arrow, which is another function, which makes it a center classification).



Center and *wall* embeddings will receive different importance scores when we combine them later. The other 6 embeddings scores for this inequality are set to values that will lead to no-op computations later in my calculations (0 in some cases; 200 in others).

To provide a second example, at the yellow arrow, we will set two embeddings but this time for a different inequality - namely for y ** 2 < np.sqrt(x) * 600. The *true*left*center* embedding (do you see why it's a *center* embedding?) and the *false*right*wall* embedding will be set. Embeddings are set to the average of the two Y values where the function switches value.

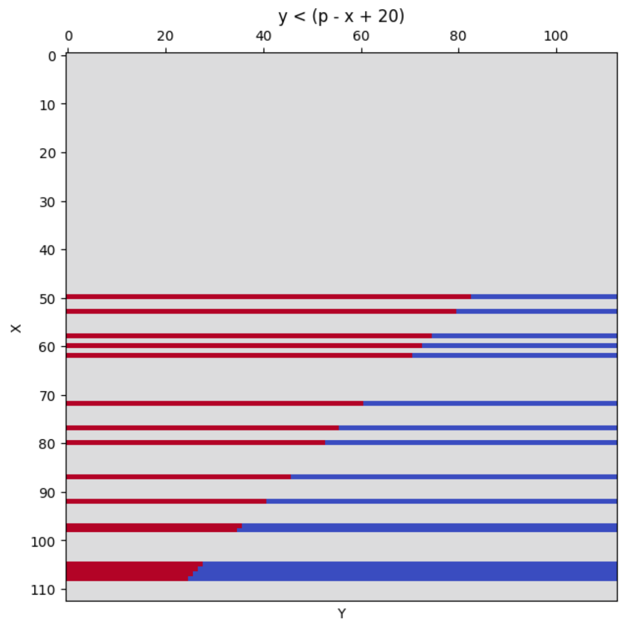
We now must think about how to combine these embeddings. When moving to the right, we perform $E_y(y) - E_x(x)$. As we saw in the example table above (partially reproduced below), the model and true label are both TRUE at 65 and above (and FALSE otherwise).

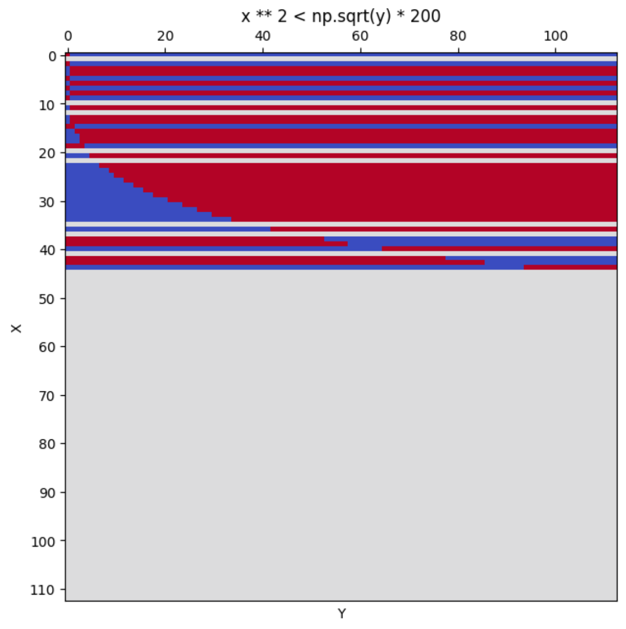
X	Ex(X)	Y	Ey(Y)	Ey(Y)-Ex(X)	model label	true label	match?
40	64.5	64	64	-0.5	FALSE	FALSE	TRUE
40	64.5	65	65	0.5	TRUE	TRUE	TRUE
40	64.5	66	66	1.5	TRUE	TRUE	TRUE
40	64.5	67	67	2.5	TRUE	TRUE	TRUE

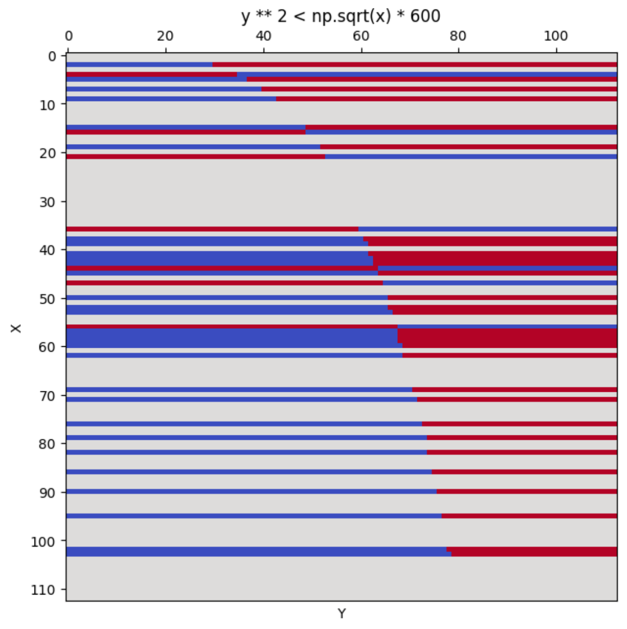
We perform the reverse when moving to the left; using $E_x(x) - E_y(y)$ gives a positive value when $E_y(y) < E_x(x)$. All of the above is viewable in my Jupyter notebook under the variable named VOM (I called it "VOM" because I picture this being the ultimate effect of an Embedding*V*O*MLP_in_W circuit, without applying the RELU, but it would have been clearer to call it something like extended_embedding).

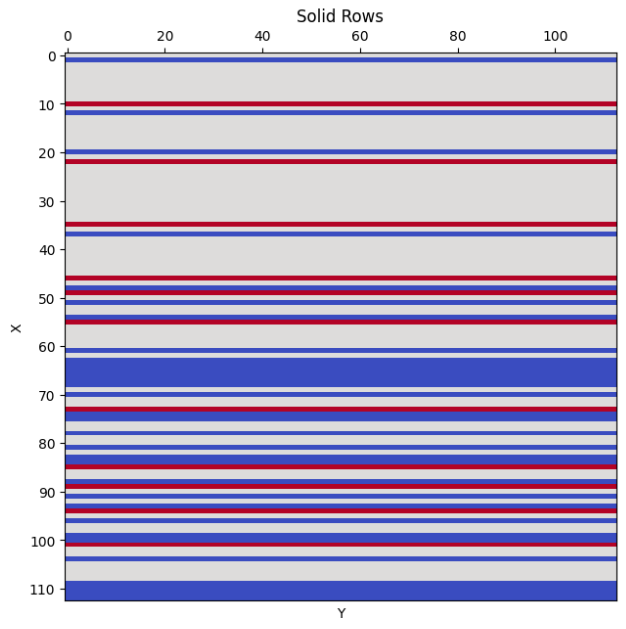
Once we've calculated these extended embeddings, we must consider how to combine them. Originally (before I made the *wall* vs *center* separation), I considered multiplying all of the *true*s by 1 and *false*s by -1, but it turns out there are issues where the *center* effects overweight the *wall* effects. Multiplying walls by a large number (100), solves this issue. *get_MLP_activations* in my Jupyter notebook implements this MLP.

We also calculate a simple extended embedding and MLP for Xs that are all true and all false. This gives us 26 total MLPs (8 for each of the 3 three functions plus these two). Images for each of these four sets are shown below (scaling of how strong these votes are is omitted).

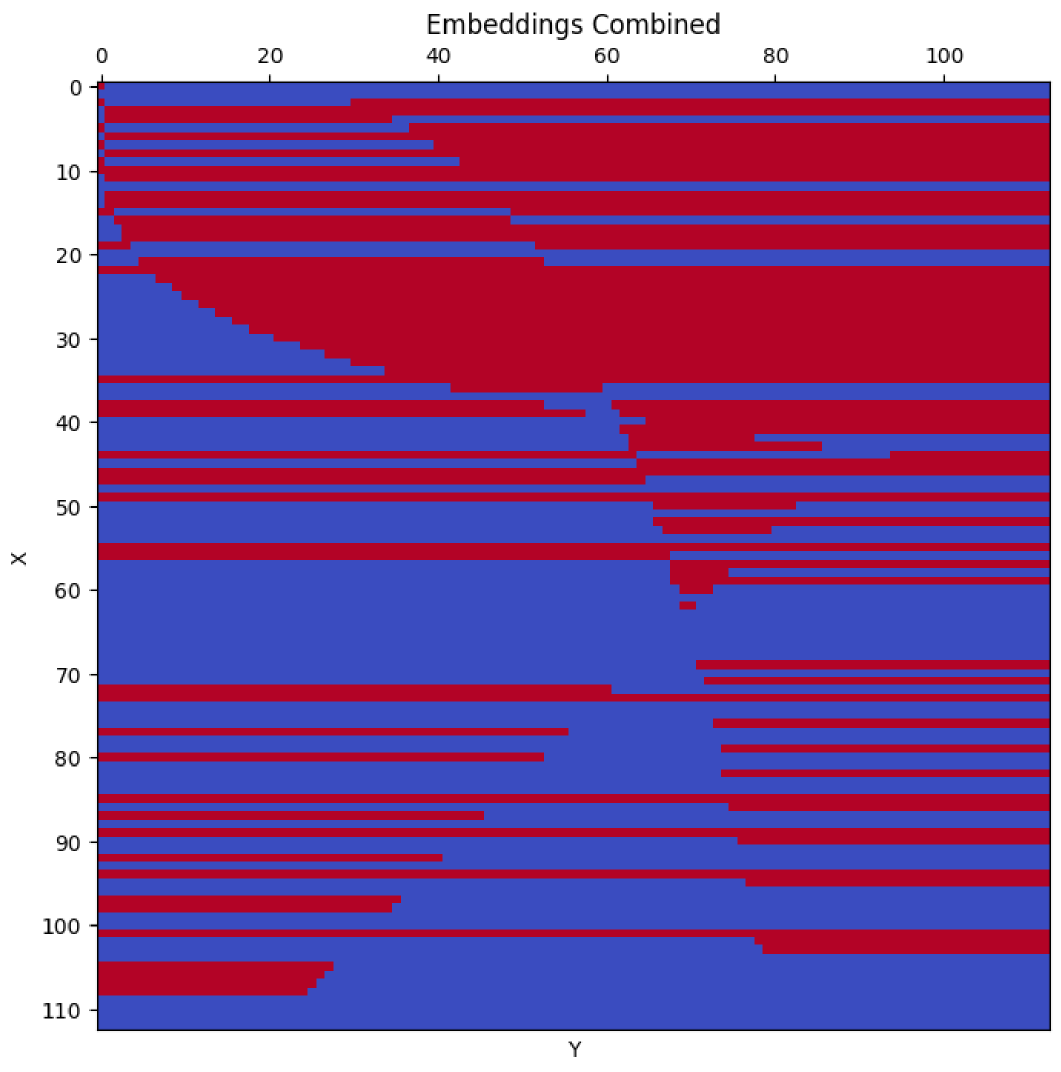








Combined these with the aforementioned weights gives the original classification function as intended. (In my Jupyter notebook, I assert that all datapoints match the original function):



To summarize what we've done so far, we've created a set of extended embeddings that when cleverly combined, can reproduce a classification function. We created 27 different embeddings. One for Y and 26 for X. Each X embedding was separately combined with the Y embedding to make 26 MLPs.

Transformer Synthesis:

We now take these extended embeddings and will use them to create a neural network with one transformer block (transformer plus MLP). The network also has embeddings (integer and positional) as well as an unembedding layer. The overall structure is copied from the original challenge, but tensor dimensions have been modified (typically decreased).

In this section, I describe how I create (synthesize) this transformer network to perfectly recreate label_fn(x, y).

My network has a dimension 30 embedding for each of the 114 possible values for X/Y. Of these 30 dimensions, 1 is used for Y, 26 are used for X, and 3 are used for the positional embedding.

The first two positional embeddings are also used to read off the prediction during the unembedding step. Note that Y and each X embedding remain in different dimensions and do not get mixed together during attention.

I would encourage you to look at the code in the Jupyter notebook, but I will provide a high level overview of the weights. I use many 0s and many identity matrices (torch.eye).

- **For W_E, the first 27 dimensions (27 x 114) are set according to the extended embeddings calculated in the previous step. This step might sound simple, but I want to highlight that the previous section was required to get these weights. In my mind, these embeddings are where most of the magic is added and will heavily be relied upon by the rest of the model.** The final 3 dimensions are all zeros as they are reserved for the positional embeddings.
- W_pos is set using an identity matrix for these 3 reserved dimensions, but it is scaled up by 10 so each position can have a large impact on the attention logit during the attention logit calculation.
- For W_K and W_Q, I have two attention heads. Attention head 1 will be responsible for copying Y's embedding into Z's dimension 0. Attention head 0 will be responsible for copying X's embedding into Z's dimensions 1-26 (inclusive). I had to play around to get this to work, but W_K and W_Q is set so that the outputted attention is [1-ε,ε/2,ε/2] for attention head 0 output location 2 (so X's embeddings can be copied into Z) and [ε/2,1-ε,ε/2] for attention head 1 output location 2 (so Y's embedding can be copied into Z). (I had to do trial and error to get this to work). (This might sound like cheating, but part of my hypothesis is that the original network is doing something similar. A circuit of attention head 2's positional embedding gets it to focus on Y more so than other attention heads. Some very rough evidence is presented here).
- W_V and W_O: W_V is set using a 26x26 identity matrix and a 1x1 identity matrix; W_O is set to be two 30x30 identity matrices; together, this allows us to copy info about X and Y, respectively into the third term. Note that since W_V doesn't touch the final 3 dimensions, no info is copied into embedding slots 27, 28, 29. We will use slots 27 and 28 for unembedding later (they also began as 0 for Z).
- W_in's first dimension will combine the Y embedding with a single X dimension. This is based on the MLP equations from the previous section. When W_in[i,0] = 1, W_in[i, i+1] = -1 and vice versa for the first 24 MLPs. The final two MLPs (solid rows of all true or all false) don't require Y. Signs are based on the *left* vs *right* split described in the previous section.
- b_in and b_out are all 0.
- W_out - MLPs marked *true* in the previous section fire positively to embedding dimension 28; MLPs marked *false* in the previous section fire positively to embedding dimension 27. The firing weight is +10 for MLPs marked *center* and +1000 for neurons marked *wall*.
- W_U - unembedding dimension 27 favors 0 (false) and discourages 1 (true). Vice versa for dimension 28.

You can run all these cells and then call eval_model() and you will see that this matches the original classification function. This is the take-away from this section, we can synthesize transformer weights so that the transformer's output perfectly matches a labelling function!

Experiments:
I ran some light-weight experiments but I think the main focus of this report should be the above syntheses.

I wanted to examine how well a linear probe works after applying attention. The aforementioned solution claims that the task is mostly solved after resid_mid. I'm curious how predictive a linear probe (hook_resid_mid[:,2,:] vs output) is for the original model as well as for my synthesized model. Note that in both cases, I compare against model output, not ground truth (this was due to time, I'm unsure which comparison is better; ideally I could have done both). (No held out set, also not sure if this was fair to do).

The original model predicts 85.2% of the input correctly from a linear probe. (I'm surprised it's this low given how well PCA works on the same residuals from the aforementioned solution).

A linear probe trained on my model predicts 75.5% of the input correctly. This makes me think my embeddings don't match with those of the original model.

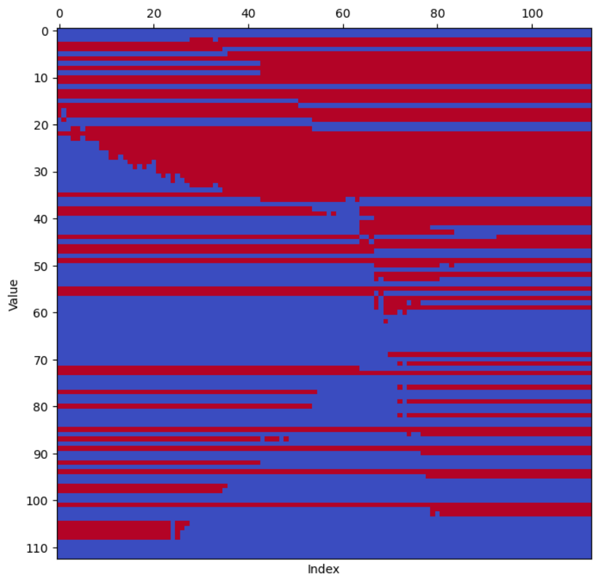
One idea I had was that some MLPs that are X_i-Y and others that are Y-X_i. I did some manipulation in the attention mechanism (Modifying W_V to subtract Y from some X_is and flipping the sign of some W_in weights) so now all functions are X_i-Y. As a result of this change, the linear probe now predicts 84.0% of the model outputs; given my model has 30 dimensions along the residual stream as opposed to 128, I think this is sufficiently close. Search for use_v2 in my Jupyter notebook to see the specific weight changes. (As mentioned earlier, the change affected W_V. I think I could have instead modified the original embeddings instead).

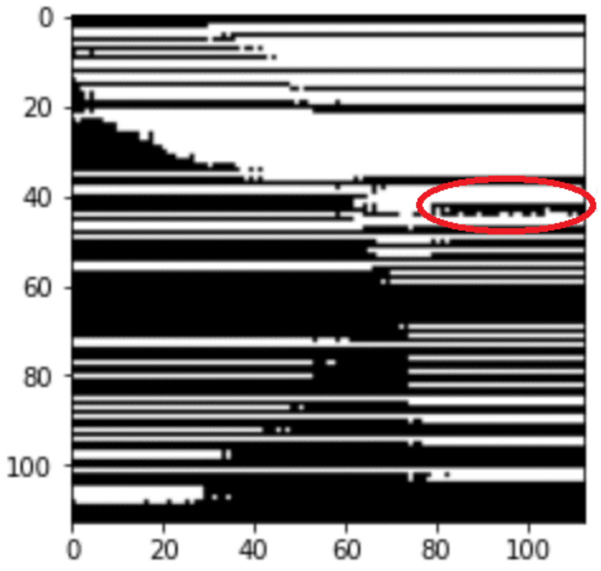
Original Model	85.2%
My Model	75.5%
My Model V2	84.0%

I'm unsure whether the original network combines X and Y in this manner. I'd also like to explore linear probes on MLP_pre for all three models (original, my model, and then my model V2) and maybe run PCA on both resid_mid and MLP_pre for all 3 models.

My take-away from this experiment is that embeddings are likely not as easily interpretable as I proposed in earlier sections.

Now an unrelated experiment. Stephen's retrospective questioned why model errors occur around the decision boundaries. My hypothesis is that it is because the model fails to sufficiently learn how to count in Y. In other words, the extended embedding for Y does not monotonically increase as it does in my syntheses. I apply random noise to the Y embedding for my synthetic transformer and get new output image. In this output image, model errors again occur near the boundary functions. This is unscientific: Qualitatively I think is a mixed bag. Many errors feel similarish, but the area I circled in red doesn't.





My noised model

Transformer Output

Future Plans & Conclusion:

I started this hackathon with a hypothesis of how a transformer network works. With one change (adding the *wall* vs *center* cut), **I was able to synthesize a set of embeddings that perfectly implements the labelling function and then use the embeddings to synthesize a transformer neural network that also perfectly implements the function.** I ran several experiments comparing my synthetic transformer to the existing transformer. I am slightly less confident that embeddings are as straight-forward as I initially expected, but I think that once I cool-off from this hackathon, my learnings will inform my continued search for the network's mechanism (it could still be similar to the modified network I tested in my experiments section).

Once I modify my hypothesis, I'd like to formalize my thinking in a document similar to this one. (It's unfortunate I don't have this nicely formalized prior to the hackathon). I hope to have this new document completed by 8/4/23.

Afterwards, I hope to explore small language models (Tiny Stories?) to perform both mechanistic interpretabilty and model synthesis. As a longer-term goal - I hope to develop code to automate mechanistic interpretability and model synthesis.

Funding Note:

I am currently on a 3 month Mechanistic Interpretability Research grant that ends in September. If you found this research interesting and are potentially interested in funding additional mechanistic interpretability research time for me, please reach out - rickgoldstein12[at]gmail[dot]com. Likewise for research positions at AI labs.