Ethan's Research Ideas

Table of Contents

1: Safety

1.1: Interpretability

1.1.1: Circuit mining via unsupervised component importance clustering

- Take some set of inputs and model predictions.
- Create some set of metrics to establish which components of the model (all the MLP layers, all the attention heads, all the layer norms, etc.) are important (contribute strongly to the output) for a particular token prediction.
- Empirically we often find that many layers of a deep, overparameterized neural network are unimportant for a particular inference.
- For each token prediction, we check each component to see how important it was.
- Do unsupervised clustering to find groups of model components which are tightly related. As in, find sets of components which are often all highly important at the same time, or all basically unimportant at the same time. These components are likely coordinating in some way to produce a specific model behavior.
- Then zoom into some particular cluster and repeat, finding clusters of tightly related components within that data subset.
- This would allow us to zoom in to particular component groups which specialize for performing certain tasks.
- Assuming we are able to find nice identifiable clusters, this would show something akin to 'feature splitting' from
  the SAE literature, where we can zoom into activations of a particular feature and find sub-features.
- Example of an interesting thing we might find: the model uses some sparse subset of components for doing math. We
  perform clustering on just math problems, and find one subset of the subset is responsible for doing subtraction, one
  for converting between currencies of different countries, one for calculating percentages, etc.
- Once we have a map of the 'behavior clusters', we can go through the sample dataset, and based on which inputs are activating in which cluster, we can try to estimate what exactly the behavior is that the group of components are coordinating to perform.
- You could do this recursively, going all the way down, making some map of nested behaviors based on the relevant components for that behavior, sort of like neuronpedia and similar projects.
- Or going backwards: given a dataset requiring some capability, we could easily just run the model on it, find out which components are important, and see where that set of importances lie relative

to the other clusters.

1.1.2: There are many methods of circuit discovery now. What do we do with the circuits?

- Has there been any work in actually using discovered circuits to 'fix' or constrain models?
- Good circuit finding works have shown examples of using the supposedly critical circuit to generate adversarial examples. In other words, to find the bugs in the algorithm the transformer implements.
- Can we use mechanistic understanding of the circuits to fix the models in a general way?
- Like ROME for any general circuit?
- Sounds pretty hard when you put it like that.
- But something in the vein of "let's use all these circuits to do something actually useful/make models safer permanently or during inference."

1.1.3: Examine yes/no arithmetic/simple math problems: what does each layer contribute to the answer across different problems, and how does the contribution of each layer change as the numerical inputs to the math problem change?

- Example: "A 6-pack of coke costs $10, a4 - packcosts$3.5. Is the 6-pack a better deal?"
- Different components will have different contributions to the yes/no logits.
- Examine different layers/components, seeing what direction each layer contributes on the yes-no spectrum.
- If we change the values in the problem, e.g., increase the cost of the cokes to $11, how does the yes-no contribution of each layer change?
- It seems like we could sort of take the gradient of each layer's yes-no contribution with respect to the inputs of the math problem, and use that to figure out specifically which operation each layer performs.
- This is only possible because of the yes/no binary answer, meaning the value of interest for each component is just a scalar, as opposed to the usual mechanistic interpretability challenge of trying to decipher what is the meaning of the gigantic vector that layer XYZ just added into the residual stream and got read in by layer ABC...

1.1.4: Training CoT models and an SAE/transcoder simultaneously to penalize cheating by identifying a hacking/dishonesty feature

- Or just training the SAE at the end and tweak the model weights afterward?
- Interesting idea: where does a feature go if we train against its presence?
- Can we track the meaning of that direction over time?

- Can we track the direction of that meaning over time?

1.1.5: Related: doing circuit discovery on models before/after some fine tuning

- Try to see if we can find ubiquitous differences, potentially relating the different circuits to the kind of fine tuning we did.
- Some circuits are probably being created during fine tuning. Do any go away, or are they just suppressed?
- Mechanistic interpretability on Anthropic's 'model organisms of misalignment' models? I don't see anyone having done this. Why does it seem kind of obvious? Is it a bad idea?

1.2: Capability Testing

1.2.1: Testing chain of thought faithfulness across models

- Models often exhibit 'unfaithful chains of thought', where based on their CoT you would expect them to arrive at a different final answer than they actually do.
- You also see that perturbing the chain of thought often results in surprisingly small effects on producing the correct answer.
- It would be interesting to transfer chains of thought across models, and see how it impacts performance, faithfulness, or error correction.
- The safety case here would be to test if models are doing some kind of steganography in the CoT that only they recognize. If this were true, we would probably not see that other models of similar capability are indifferent to perturbations in the chain of thought they are given.
- It would also just be interesting to see performance differences. Do models become highly specified to their own style of thought?
- Alternatively, we could train models on tasks which encourage/require some form of steganography in the chain of thought, as a model organism, and try the other experiments on it.
    - Maybe an LLM vs LLM game where each model can see the other's chain of thought? They have to learn to use their own chain of thought while deceiving the other.
    - Could result in a stalemate if the models simultaneously learn to read the other's steganography scheme.

1.2.2: How good are models at noticing confusion?

- An important part of the whole agentic language model, feedback from the environment thing is noticing when the outputs from the environment are unexpected or confusing.

- A very common failure mode of these models is when they don't pick up on when something is amiss, something that would challenge their assumptions and cause them to rethink their plans.
- Language models are trained as language predictors, so one might expect them to be very good at noticing unlikely tokens, but the only info a model really has available is in the CoT, which logit outputs are not.
- The project would be to test how good models are at identifying outputs which were unlikely, given the model's understanding of the environment generating the data.
- Could use simple programs which print to the terminal, showing the model the output, then simply asking it to say how unlikely it thinks that output was.
- We could then switch out the proper program output with an incorrect one, and see if the model is able to identify that the output is unlikely, or does it succumb to hindsight bias and say that everything it already knows was likely a priori.
- Could test with models with different training recipes (one-shot, reasoning models, etc.)

2: Capabilities

2.1: Chain of Thought

2.1.1: Model with invisible 'thinking' tokens trained via RL, with rewards based on supervised token accuracy

- Thinktoks was a project I started but never saw results on due to training code being very complicated.
- During ordinary pretraining, the model autoregressively outputs tokens on which we calculate cross-entropy against (normal supervised loss) the correct token for each sequence position.
- The idea was to add a bunch of extra tokens to the model's vocab.
- These tokens would be *ignored* when they are output during training (when calculating the loss that is). We continue spitting out tokens autoregressively (like during model inference) until we have produced as many 'real' tokens as were in the output sequence, ignoring all the extra 'thinking' tokens.
- During training, the model will receive no supervised feedback whatsoever when a thinking token is output.
- Instead, we include another loss term to the total loss for a sequence. The loss of a particular invisible token is a function of the ordinary, supervised loss for all the ordinary tokens which were output later.

- So the reward signal for normal tokens is simply to correctly predict the next token, like a normal model, except there are invisible tokens in the context which weren't part of the input sequence, but can be attended to for next token prediction.
- The reward signal for the invisible tokens is a function of the loss of all following real tokens.
- So we encourage the model to output normal tokens which are the same as the actual next token, and to output invisible tokens which increase the chance of outputting the correct prediction for the next token.
- It would also be probably necessary to apply a fixed cost for outputting invisible tokens to get it to eventually output real token predictions instead of "thinking forever."
- The idea is to allow the model to think, but not out loud. To be able to output intermediate results of unconstrained semantic meaning, and to do this for as many tokens as required.
- Language models contain in their weights algorithms which solve the problems which it is presented with in the task of next token text prediction (IOI, arithmetic, etc). These algorithms are however limited in number of steps by the number of layers in the model. (https://arxiv.org/abs/2210.10749)
- The idea of hidden tokens is to allow them to discover and learn arbitrary length (in time or space) algorithms to apply to the problems present in NTP.
- The current generation of reasoning models are doing reasoning using invisible tokens trained via RL. The difference is that they are trained to reason in post-training, and using normal tokens which are just not checked. This makes the largest issue finding supervision for the RL, which is why the recent reasoning models are mainly improved in just math and code, where answers are verifiable and questions are synthesizable.
- This would allow models to learn to reason during pretraining and all other places, because the RL reward is just next token prediction accuracy. This method can be applied on top of the other reasoning tech too.
- I have recently found out about the Coconut paper from December of 2024. It describes something similar to this. They also use augmented tokens (the same tokens, used in a different way) used for thinking, which are ignored when calculating supervised loss. They differ in that they do not expand the dictionary or unembed these special tokens, and simply leave thought tokens as continuous when they start the next token generation. They also use `<thinking>` tags to enclose a thinking segment of a set number of augmented tokens.
    - They make no mention of RL. The paper made me realize that the RL is not necessary at all. The gradient for the supervised loss will propagate through the thought tokens naturally, as the normal supervised token positions will attend to and read info from the thought token. So yeah.
    - The main difference here is that they are still only doing reasoning training as part of post-training, and still only using chain of thought in post-training. The method I propose is a deeper modification to the fundamental operation of the transformer, and involves reasoning

    - which happens on the most fundamental level, including during pretraining, post-training, and basically any time the model is outputting tokens.

2.1.2: RL on chain of thought in game playing domains

- Current LLM reasoning-through-RL approaches require verifiable domains.
- Therefore, the main domains where improvements have been the largest for reasoning models are math and code.
- Games are also a verifiable domain (the one who won probably played better).
- Could you do RL on chain of thought in an adversarial game setting?
- With an LLM, you could even have the training include multiple different games: Go, chess, etc. all at the same time.
- If it works, this could potentially open the door to do reasoning-RL on more domains. Domains in which solutions are not objectively verifiable, but where you can objectively say which of two solutions is better.
- Risk, Mafia, etc. involve conversation, negotiation, or deception as central mechanics. How well can models learn to outwit other models in conversation?
    - Lol, maybe they discover and deploy jailbreak sequences to cause other models to go haywire.

2.1.3: Or just ask the models what they should have done

- This is part of why human learning is so much more efficient than SGD. Humans consciously (rather than via stupid first-order gradient-based updates) incorporate experience into our future decision making process by first recognizing they messed up, then asking themselves: "Was there a proper line of reasoning which I should have been able to recognize before I messed up that would have led me to not mess up? If so, what was it?" Then they think hard about what they should've thought, and sometimes say: "Yeah, in this situation it should have been obvious that the correct approach was X and that Y is a mistake," and that gets incorporated into their future decision process.
- In a similar vein, what if we just instruct the agent to occasionally, upon receiving some feedback from the environment, recognize if they made a mistake, and just ask them to output what they should've said in the first place.
- I guess the idea would be to then RL the chain of thought of the unexperienced model to output what the more experienced model says they should have done in hindsight.
    - Or even just train a compact adapter like a LoRA finetune and then you could have a bunch of these adapters for different tasks. Like if you need it to code, plug in the coding assistant adapter, if you need it to be a personal assistant, put on the assistant adapter, etc.

- Can models do this? It would be a huge unlock if they can. It would basically allow the models to use their general common sense reasoning + environment feedback to recognize when they have messed up or are not on the right path, generating reward signals and specific, tailored fixes to their own behavior.
- This totally eliminates the need for verifiable domains.
- You would actually need an unchanging reference model to make the feedback judgements, otherwise you just get models that constantly upvote themselves.
- Need to read more about constitutional AI.

2.1.4: Training CoT models with distorted reasoning traces to enable robust error correction

2.2: New Modalities

2.2.1: Transformer models for program binary decompilation

- While attending to the whole input binary file, autoregressively output the source code.
- For training data, I imagine we could use packages available via package managers (pip, pacman, apt, flatpak). These often work by downloading source code from GitHub or similar and then compiling locally.
- This gives us convenient access to a huge database of real-world programs in both source code and compiled format.
- Probably would want to output a serialized AST or something instead of normally tokenized text.
- But it would be cool to see if it can figure out likely variable names and stuff just from the code structure.

2.2.2: Transformer models for 3D object generation

- Fine-tune instruct models on 3D object files associated with object descriptions.
  - What kind of 3D object file?
    - Most object files contain a set of unordered vertices and an unordered set of faces.
    - Maybe we just do loss based on if the predicted vertex/face is anywhere in the set?
- Can we then get it to generate object files from user descriptions?
- Actually, can you fine-tune instruct models on new modalities of data like this and get it to follow instructions properly? I do not know.
  - Might be worth a paper in its own right.

3: Unstructured questions:

How do different modalities 'mesh' in the 'cognition' of the transformer?

- OpenAI recently tried applying GPT-4o to the task of protein design. Since language models can handle a much higher bandwidth of data, they can literally just 'read the amino acids' in a way that is impractical for humans. In the article I read, it suggests that they simply trained it on AA sequences, then asked it questions and for suggestions in natural language, and that this was somewhat fruitful (for a company that was designing new proteins to make normal cells turn back into stem cells).
- Is this a generally applicable paradigm for extracting patterns from opaque data? If a language model has understanding (the capability to next-token-predict) some type of data, as well as language understanding, is it able to translate its understanding from the one modality to normal language?
- This seems unlikely in the strong case, but obviously happens to some degree. It seems plausible that this failure of understanding across domains could explain poor visual understanding in LLMs, as well as lack of chain of thought faithfulness. Vision/math understanding isn't properly being transferred.
- Other examples of 'opaque data' could be brain activity, obfuscated code, encrypted data, etc. Can you just train on one and then talk about it with the model?
- Could investigate how this mesh happens, or how to increase the degree of meshing. Computer use agents still forget how to click on buttons.

Mechanistic interpretability on reasoning models

- For example, agentic-type reasoning models have to sometimes realize "this approach isn't working, let me try something else." Can we discover the circuit that triggers this? Is it absent in non-reasoning models?
- A reasoning model also has to choose when to stop thinking and start 'speaking'. Can the responsible circuit be discovered? Can it be intervened upon to make a model think more/less than it normally would?
- Do the 'internal meanings' of some reasoning-critical tokens change?