Compiled Transformers as a Laboratory for Interpretability

David Lindner, János Kramár, Matthew Rahtz, Tom McGrath, Vladimir Mikulik

April 12, 2023
One reason interpretability is hard is that there is no ground truth information.
What if we could create situations where we do have a ground truth?

Neural Network \[\rightarrow\] Interpretability \[\rightarrow\] Explanation

Known Mechanism \[\leftarrow\] Is the explanation correct?
Introducing Tracr: A Transformer Compiler for RASP

Known Mechanism → Neural Network
Plan for today

1. Building a **compiler** for transformer models

2. Studying **superposition** in compiled models
Plan for today

1. Building a compiler for transformer models

2. Studying superposition in compiled models
Hand-coding weights is useful but not scalable

- We can **hand-code** weights to build **ground truth models**
- Very good **measure** of how good we understand a model
- **Difficult to scale** to more complex/bigger models
- This approach is like **programming in byte-code**

Tracr works analogously to how we would translate a programming language into executable code.
Tracr translates human readable code into transformer model weights in three steps

1. Human readable code in domain-specific language
2. Basis independent representation of vector spaces and transformers
3. Neural network weights
RASP is a symbolic programming language to describe transformer computations.

Arbitrary element-wise functions

“Select-aggregate” operations

(f())

MLP layers

Attention layers

RASP = “Restricted Access Sequence Programming”

An example RASP program

```
seq := ["a", "x", "b", "x", "c"]
tokens(seq) = ["a", "x", "b", "x", "c"]
indices(seq) = [0, 1, 2, 3, 4]
is_x(seq) = [0, 1, 0, 1, 0]
prevs(seq) = [[1, 0, 0, 0, 0],
               [1, 1, 0, 0, 0],
               [1, 1, 1, 0, 0],
               [1, 1, 1, 1, 0],
               [1, 1, 1, 1, 1]]
frac_prev(seq) = [0, 1/2, 1/3, 2/4, 2/5]
```
Translating a RASP program into a craft transformer

**Step 1:** Create computational graph

**Step 2:** Infer inputs/outputs

**Step 3:** Create model components

**Step 4:** Assign components to layers

**Step 5:** Assemble craft model

\[
\text{is}_x = (\text{tokens} == \text{"x"})
\]

\[
\text{prevs} = \text{select}(\text{indices}, \text{indices}, \leq)
\]

\[
\text{frac}_\text{prev} = \text{aggregate}(\text{prevs}, \text{is}_x)
\]
We implement MLP layers to approximate arbitrary pointwise functions.

For categorical variables:
MLP = Lookup table

For numerical variables:
Approximate using ReLU

We can ensure this is correct on a discrete set of possible inputs.
Attention heads can implement arbitrary selectors with categorical inputs

- We can use a low softmax temperature to make attention patterns binary
- But what if we don’t want to attend to anything?
  - We add a beginning of sequence token that we can always attend to
  - Anecdotally, it seems like real transformers also do this!

\[
\text{select}(\text{indices, indices, } \leq) \rightarrow W_Q^T W_K \\
\text{aggregate}(\text{prevs, is\_x}) \rightarrow W_O^T W_V
\]
A craft model can be directly mapped to any standard transformer architecture.

Abstract representation of a transformer that makes it easier to reason about vector spaces. Can be mapped to any GPT-like transformer implementation.

We primarily support a standard haiku transformer implementation.
Tracr can compile a range of meaningful programs, but it is not fully general

We can implement programs to:
- Count tokens and compute histograms
- Detect all occurrences of a patterns
- Sort the input sequence
- Check balanced parentheses (Dyck-n)
- ...

Limitations of RASP
- Binary attention patterns
- Designed to model algorithmic tasks and not probabilistic tasks
- Programs still relatively close to transformer architecture

Limitations of Tracr
- Resulting models are large and inefficient
- Many possible optimization missing
- Some advanced RASP features not supported
We can now compile RASP programs!
Plan for today

1. Building a **compiler** for transformer models
2. Studying **superposition** in compiled models
Plan for today

1. Building a **compiler** for transformer models

2. Studying **superposition** in compiled models
The superposition hypothesis

**Observation 1:** Sometimes neurons correspond to clearly interpretable features.

**Observation 2:** Sometimes neurons seem to represent multiple interpretable features.

**Observation 3:** A linear representation can approximately embed exponentially more features than it has dimensions.
The superposition hypothesis

Neural networks simulate larger networks with disentangled features

These hypothetical features are projected into the actual network using superposition

This results in polysemanticity when looking at single neurons.

Superposition occurs in toy models


\[ h = Wx \]
\[ x' = \text{ReLU}(W^T h + b) \]
\[ x' = \text{ReLU}(W^T W x + b) \]

\[ L = \sum_x \sum_i \frac{I_i(x_i - x_i')^2}{2} \]
We expect superposition to occur, if we “compress” Tracr models to be more efficient

In toy models we see superposition if
1. Features are sparse
2. Some features are more important than others
3. The model has to use fewer dimensions than features

In Tracr models
1. Features are sparse
2. Some features are more important for the computation
3. Can we “compress” the model to use fewer dimensions?

Motivation
a. Learn something about superposition in more realistic models
b. Make Tracr models more naturalistic
We linearly compress the model’s residual stream

Train (only) $W \in \mathbb{R}^{D \times d}$ to minimize:

$\mathcal{L}(W, x) = \mathcal{L}_{\text{out}}(W, x) + \mathcal{L}_{\text{layer}}(W, x)$

$\mathcal{L}_{\text{out}}(W, x) = \text{loss}(f(x), \hat{f}_W(x))$

minimize output loss

$\mathcal{L}_{\text{layer}}(W, x) = \sum_{\text{layer } i} (h_i(x) - \hat{h}_{W,i}(x))^2$

implement the same computation

The model should minimize loss under the constraint that it implements the same computation.
The embeddings show superposition that is qualitatively different from PCA embeddings.
Which features will be stored in superposition?

Feature importance

Feature Density

Linear Independence
Which features will be stored in superposition?

Feature importance

+ Feature Density

+ Linear Independence

Open question:
Can we find a more predictive description of which features will be stored in superposition?
Plan for today

1. Building a **compiler** for transformer models

2. Studying **superposition** in compiled models
The future for Tracr and manual transformers

1. Can we use Tracr to create **evaluation benchmarks** for interpretability tools?

2. Can we **revert superposition** in Tracr models? (e.g., sparse coding, dictionary learning)

3. Can we use Tracr to **manually replace** model components that we (think we) understand?
Tracr is available open-source!

https://github.com/deepmind/tracr

https://arxiv.org/abs/2301.05062

Thank you!