Explainable Transfromers
(Learning Transformer programs)

Enrico Santi
santi.enrico@spes.uniud.it

University of Udine

February 2024

Presentation overview

1. Introduction
2. Thinking like Tranformers
3. Learning Transformers Programs
4. Conclusions

Introduction

Explainable AI in a nutshell

Models in AI/ML can be subdivided into :

- white boxes (e.g. trees, kNN, logistic regression...)
- black boxes (deep neural networks and complex models)

XAI deals with the black boxes, trying to improve some of their aspects (i.e. Transparency, Interpretability and Trustworthiness), shifting them towards grey/white boxes.

Why do we care ?

Explainable AI in a nutshell

Models in AI/ML can be subdivided into :

- white boxes (e.g. trees, kNN, logistic regression...)
- black boxes (deep neural networks and complex models)

XAI deals with the black boxes, trying to improve some of their aspects (i.e. Transparency, Interpretability and Trustworthiness), shifting them towards grey/white boxes.

Why do we care ? $\rightarrow$ in many critical applications we can't leverage a quasi-perfect black box model as we'd leverage a 100% correct black box, in addition, we'd like to be provided some information on the answer.

Focusing on Transformers

Transformers are a popular N.N. architecture representing the state-of-the-art in NLP and are also being used in other domains (why ? $\rightarrow$ because they work).

Focusing on Transformers

Transformers are a popular N.N. architecture representing the state-of-the-art in NLP and are also being used in other domains (why ? $\rightarrow$ because they work).

$\downarrow$

Problem : they are huge black boxes, we have little knowledge on how they actually work and why some answer are given.

Focusing on Transformers

Transformers are a popular N.N. architecture representing the state-of-the-art in NLP and are also being used in other domains (why ? $\rightarrow$ because they work).

$\downarrow$

Problem : they are huge black boxes, we have little knowledge on how they actually work and why some answer are given.

$\downarrow$

When a system using them doesn't work correctly good luck in finding the cause. In addition we can't leverage them in critical systems.

What has been done before

- The Post-hoc approaches implemented didn't provide a complete description of the behaviour.

What has been done before

- The Post-hoc approaches implemented didn't provide a complete description of the behaviour.
- *Thinking like Tranformers* : A paper introducing a computational model for Transformer encoders and a programming language (RASP) to build them (see `https://www.youtube.com/watch?v=H-x_upYg-JY`).
  In addition a research line has been focusing on reverse engineering Transformers by characterizing their components (attention heads, feed forward layers...) into "circuits".

What has been done before

- The Post-hoc approaches implemented didn't provide a complete description of the behaviour.
- *Thinking like Tranformers* : A paper introducing a computational model for Transformer encoders and a programming language (RASP) to build them (see `https://www.youtube.com/watch?v=H-x_upYg-JY`).
  In addition a research line has been focusing on reverse engineering Transformers by characterizing their components (attention heads, feed forward layers...) into "circuits".

$$\downarrow$$

We will build on this latter approach.

What has been done before

- The Post-hoc approaches implemented didn't provide a complete description of the behaviour.
- *Thinking like Tranformers* : A paper introducing a computational model for Transformer encoders and a programming language (RASP) to build them (see `https://www.youtube.com/watch?v=H-x_upYg-JY`).
  In addition a research line has been focusing on reverse engineering Transformers by characterizing their components (attention heads, feed forward layers...) into "circuits".

↓

We will build on this latter approach.

Bonus : a line of research is exploring the connection between formal languages and Transformers, aiming to formally characterize their expressivity.

Thinking like Tranformers

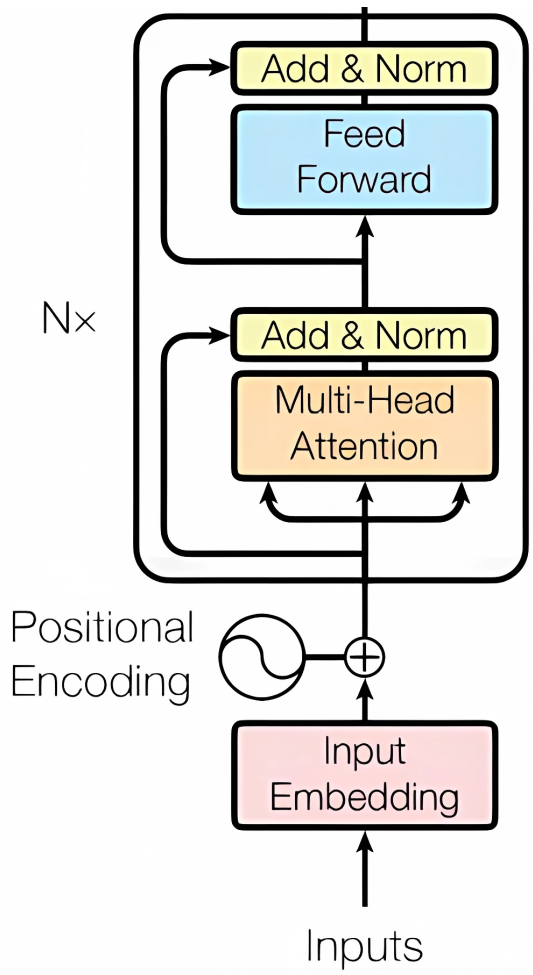
in less than a nutshell

Thinking like Tranformers

A Transformer computes a function from sequences to sequences. A RASP computation takes as input sequence of length n and returns a sequence. In addition the computation involves only sequences of length n and $n \times n$ matrices.

↓

For each element in the encoder there's a relative function in RASP, which can take as input such sequences or matrices, abstracting such elements into primitives.



Thinking like Tranformers (cont.)

Positional encoding	$\rightarrow$	The computation takes two sequences, the input and a sequence : $[0..n]$.
Feed forward layers	$\rightarrow$	They are universal function approximations, we use the basic operations (element-wise) on two sequences to replace them.
Attention head	$\rightarrow$	Obtained using the *Selection* and *Aggregate* primitives.

$$sel([0,1,2],[1,2,3],<) = \begin{bmatrix} \mathbf{T} & F & F \\ \mathbf{T} & \mathbf{T} & F \\ \mathbf{T} & \mathbf{T} & \mathbf{T} \end{bmatrix} \quad agg(\begin{bmatrix} \mathbf{T} & F & F \\ \mathbf{T} & \mathbf{T} & F \\ \mathbf{T} & \mathbf{T} & \mathbf{T} \end{bmatrix}, [10,20,30]) = [10,15,20]$$

Thinking like Tranformers (cont.)

Positional encoding	$\rightarrow$	The computation takes two sequences, the input and a sequence : $[0..n]$.
Feed forward layers	$\rightarrow$	They are universal function approximations, we use the basic operations (element-wise) on two sequences to replace them.
Attention head	$\rightarrow$	Obtained using the *Selection* and *Aggregate* primitives.

$$sel([0,1,2],[1,2,3],<) = \begin{bmatrix} \mathbf{T} & F & F \\ \mathbf{T} & \mathbf{T} & F \\ \mathbf{T} & \mathbf{T} & \mathbf{T} \end{bmatrix} \quad agg(\begin{bmatrix} \mathbf{T} & F & F \\ \mathbf{T} & \mathbf{T} & F \\ \mathbf{T} & \mathbf{T} & \mathbf{T} \end{bmatrix}, [10,20,30]) = [10,15,20]$$

Aren't we adding too much expressiveness ?

Thinking like Tranformers (cont.)

Positional encoding	$\rightarrow$	The computation takes two sequences, the input and a sequence : $[0..n]$.
Feed forward layers	$\rightarrow$	They are universal function approximations, we use the basic operations (element-wise) on two sequences to replace them.
Attention head	$\rightarrow$	Obtained using the *Selection* and *Aggregate* primitives.

$$sel([0,1,2],[1,2,3],<) = \begin{bmatrix} \mathbf{T} & F & F \\ \mathbf{T} & \mathbf{T} & F \\ \mathbf{T} & \mathbf{T} & \mathbf{T} \end{bmatrix} \quad agg(\begin{bmatrix} \mathbf{T} & F & F \\ \mathbf{T} & \mathbf{T} & F \\ \mathbf{T} & \mathbf{T} & \mathbf{T} \end{bmatrix}, [10,20,30]) = [10,15,20]$$

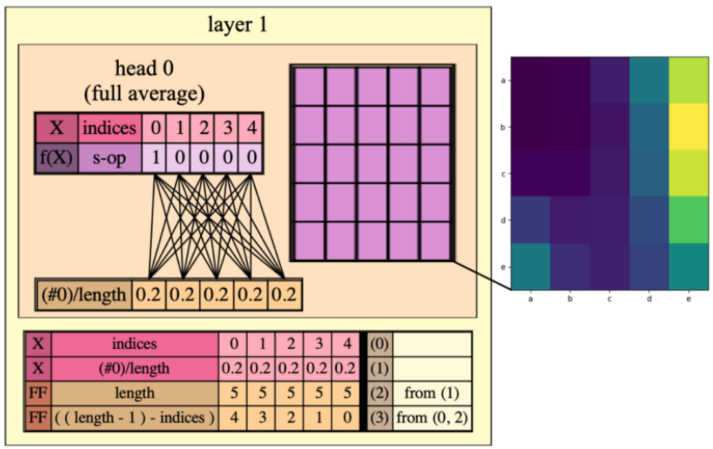
Aren't we adding too much expressiveness ?
No, the information flow is the same as within an encoder, input dependent loops are NOT allowed (not Turing complete).

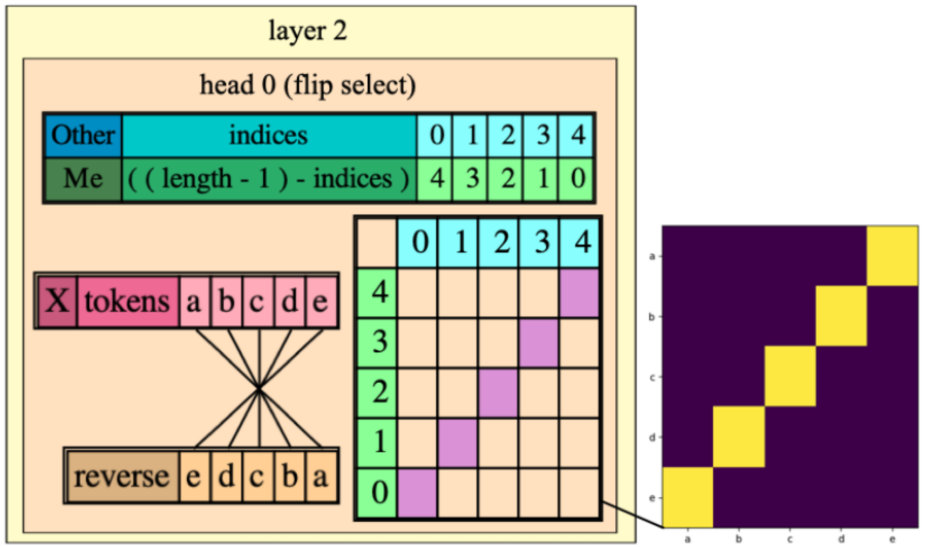
Thinking like Tranformers (cont.)

Once compiled, a RASP program should give us an encoder architecture sufficiently small (still an upper bound relative to a classical Transformer) to solve the desired task.

- To evaluate the tightness of the upper bound, smaller encoders are trained and the performance are then compared.
- In addition, the selector patters should be relevant (i.e. close to the attention matrices).

```
1  opp_index = length - indices - 1;
2  flip = select(indices, opp_index,==);
3  reverse = aggregate(flip, tokens);
```





But where are the weights ?

The original paper doesn't focus on the weights generation given the RASP program.

↓

None the less their generation is possible.

↓

Indeed a 2023 paper [4] with some small tweaks to RASP was able to translating RASP to model weights (and the relative architecture). The name of such compiler is **Tracr**.

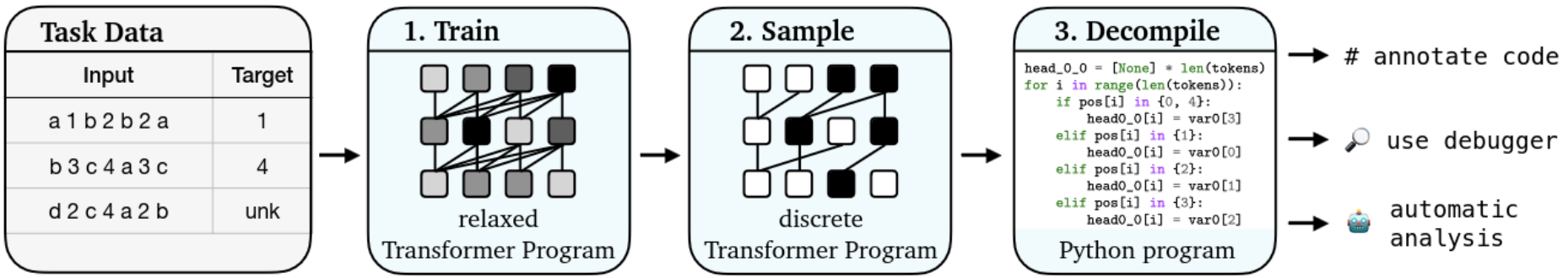
Learning Transformers Programs

Overview

RASP has been introduced to be able to go from human understandable code to an Encoder.

The idea of this paper, on the other hand is to train a Transformer (with some additional constraints) and then decompile it into a human readable RASP program. Thus, the resulting programs will be coming from Learned Tranformers.

Lastly it will be shown that even with the additional constraints (restricting the Transformers to be within an interpretable subspace) the Transformers programs will achieve similar performances with respect to their unconstrained Transformer counterparts.



What are Transformers - informally

To see how the constraints will limit the Encoder we will first need to have an idea of how Tracr informally sees the encoder itself.

Encoder : A series of layers (both attention and feed forward) which read the *residual stream* (i.e. the sequence of token embeddings, $x_0..x_L \in \mathbb{R}^{N \times d}$) computed by each previous layer ($0..L$), calculate a function and then rewrite on the residual stream.

What are Transformers - informally

To see how the constraints will limit the Encoder we will first need to have an idea of how Tracr informally sees the encoder itself.

Encoder : A series of layers (both attention and feed forward) which read the *residual stream* (i.e. the sequence of token embeddings, $x_0..x_L \in \mathbb{R}^{N \times d}$) computed by each previous layer ($0..L$), calculate a function and then rewrite on the residual stream.

The residual stream is the sum of the outputs of all the previous layers and the original embedding. Think it as a communication channel since it doesn't do any processing itself.

The read and write operations though are done via two projection matrices, respectively $W_{in} \in \mathbb{R}^{d \times d_h}$ and $W_{out} \in \mathbb{R}^{d_h \times d}$

The constraints

The first constraint regards the residual stream :

Each variable is written in an orthogonal subspace of the stream (i.e. each layer stores the output of its computation in a new set of variables).

Each layer will then read a fixed set of variables, to do so, the projection matrices are parameterized via a one-hot vector over the available variables.

The constraints

The first constraint regards the residual stream :

Each variable is written in an orthogonal subspace of the stream (i.e. each layer stores the output of its computation in a new set of variables).

Each layer will then read a fixed set of variables, to do so, the projection matrices are parameterized via a one-hot vector over the available variables.

Example : Suppose the residual stream encodes m categorical variables (of with cardinality k), resulting in embeddings $x_i \in \{0,1\}^{N \times mk}$. Then each projection matrix $W \in R^{mk \times k}$ is defined as $W = [\pi_1 I_k; ...; \pi_m I_k]^T$, in which only one π_i is equal to 1. So, each attention head is associated with three gate vectors, π_K , π_Q, π_V, defining the key, query, and value variables.

The constraints (cont.)

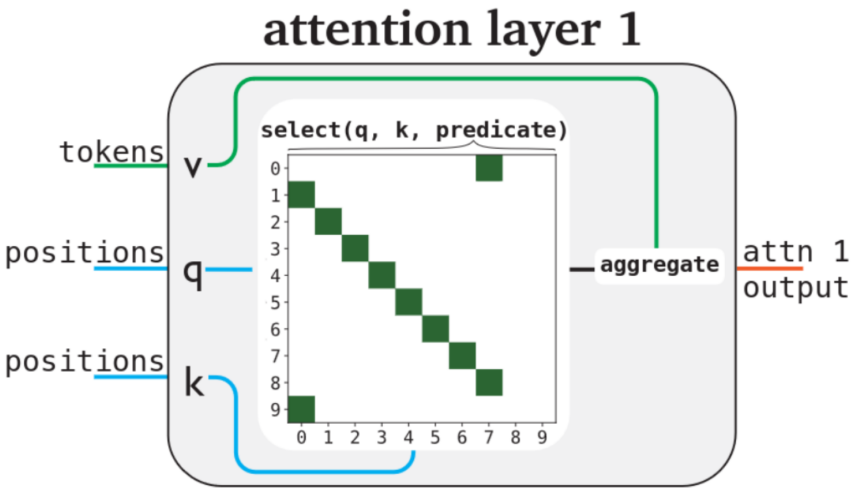
The second constraint is necessary to implement each layer as an interpretable mapping between inputs and outputs.

For each module in a layer a mapping has to be defined (i.e. we need to implement the RASP primitives).

As an example we provide only the computation of the categorical attention module :

In RASP this operation is defined via a boolean predicate, which maps every combination of key and query to a $0, 1$ value. A predicate is "learned" associating each attention head with a one-to-one mapping between query values and key values. Suppose all variables have cardinality k, each attention head is associated with a binary predicate matrix, $W_{predicate} \in \{0, 1\}^{k \times k}$, with the constraint that each row sums to one.

The constraints (cont.)




```
def predicate_1(q_position, k_position):
    if q_position in {0, 8}:
        return k_position == 7
    if q_position in {1, 9}:
        return k_position == 0
    if q_position in {2}:
        return k_position == 1
    if q_position in {3}:
        return k_position == 2
    if q_position in {4}:
        return k_position == 3
    if q_position in {5}:
        return k_position == 4
    if q_position in {6}:
        return k_position == 5
    if q_position in {7}:
        return k_position == 6

attn_1_pattern = select_closest(
    positions, positions, predicate_1)
attn_1_outputs = aggregate(attn_1_pattern, tokens)
```

Some results - the tasks

Some of the tasks on which Transformer programs have been tested :

- **In context learning problem** : Given a sequence of alternating letters and numbers ending with a letter, return *unkonw* if the last character wasn't present before in the sequence, else return the number appearing after the letter.
- **Most-freq** : Given a sequence of letters, return the unique input tokens in descending frequency order (position to break ties).
- **Dyck-2** : Given a sequence of characters in $(,), \{, \}$ return a string of characters P, T, F for which in position i we have P if the improper prefix is a dyck-2 string, P if the proper prefix is a dyck-2 string, F else.
- **Name entity recognition**.

Some results - the details

For the tasks presented, the following results were obtained by the best performing Transformers Programs models :

Task	max inp. size	no. layers	no. heads	MLPs	Acc.
In context l.	10	2	1	0	$\sim 100\%$
Most-freq	8	3	8	4	75.69%
Dyck-2	16	3	8	2	99.09%
Name entity r.	32	2	8	N.A.	81.0%

Performances seem on the same level as Standard transformers of a comparable size.

Some results - the details

For the tasks presented, the following results were obtained by the best performing Transformers Programs models :

Task	max inp. size	no. layers	no. heads	MLPs	Acc.
In context l.	10	2	1	0	$\sim 100\%$
Most-freq	8	3	8	4	75.69%
Dyck-2	16	3	8	2	99.09%
Name entity r.	32	2	8	N.A.	81.0%

Performances seem on the same level as Standard transformers of a comparable size.

Look at the input sizes though !

Some results - In contex learning in depth

```
def predicate_1(q_position, k_position):
    if q_position in {0, 8}:
        return k_position == 7
    if q_position in {1, 9}:
        return k_position == 0
    if q_position in {2}:
        return k_position == 1
    if q_position in {3}:
        return k_position == 2
    if q_position in {4}:
        return k_position == 3
    if q_position in {5}:
        return k_position == 4
    if q_position in {6}:
        return k_position == 5
    if q_position in {7}:
        return k_position == 6

attn_1_pattern = select_closest(
    positions, positions, predicate_1)
attn_1_outputs = aggregate(attn_1_pattern, tokens)
```

```
def predicate_2(token, attn_1_output):
    if token in {
        "<pad>", "0", "1", "2", "3", "<s>"
    }:
        return attn_1_output == "<pad>"
    if token in {"a"}:
        return attn_1_output == "a"
    if token in {"b"}:
        return attn_1_output == "b"
    if token in {"c"}:
        return attn_1_output == "c"
    if token in {"d"}:
        return attn_1_output == "d"

attn_2_pattern = select_closest(
    attn_1_outputs, tokens, predicate_2)
attn_2_outputs = aggregate(attn_2_pattern,
                           tokens)
```

Some results - In contex learning in depth

Regarding the code in the previous slide :

- The first attention head : given the query and key (in terms of the position of the variable to look for) learns to associate to such position the key of the previous position, the content of the *token* representing variables is written as the value.
- The second attention head : given the previous output as the keys, and the input tokens as queries, checks if the query is a letter (the sequence ends with a letter) and is such case “searches” in the previous output the token matching such letter.

Conclusions

Conclusions

The method presented allows generate from trained (and constrained) Transfromer models into discrete human readable programs.
Such constrained trasformers are still capable of learning the solution to many tasks.

Nothing is perfect though.

Conclusions - toy world problem ?

The Trasformer programs/constrained Trasformed tested struggled with longer inputs (may be the toy world problem many AI algorithms faced).
In the paper this is supposed to be caused by optimization issues and not by the expressivity of the framework.

↓

Why this supposition ? Because they were able to write RASP code (a Trasformer program) which when complied (in a constrained transformer) achieved high accuracy.

More studies probably will follow.

Conclusions - Interpretability ?

Discrete Transformer programs are easier to interpret than classic Transformers :

- Debuggin for certain problem instances becomes easier.
- Automatic analysis of the program can be done (e.g. formal verification).

Conclusions - Interpretability ?

Discrete Transformer programs are easier to interpret than classic Transformers :

- Debuggin for certain problem instances becomes easier.
- Automatic analysis of the program can be done (e.g. formal verification).

But are *humanly* interpretable ?

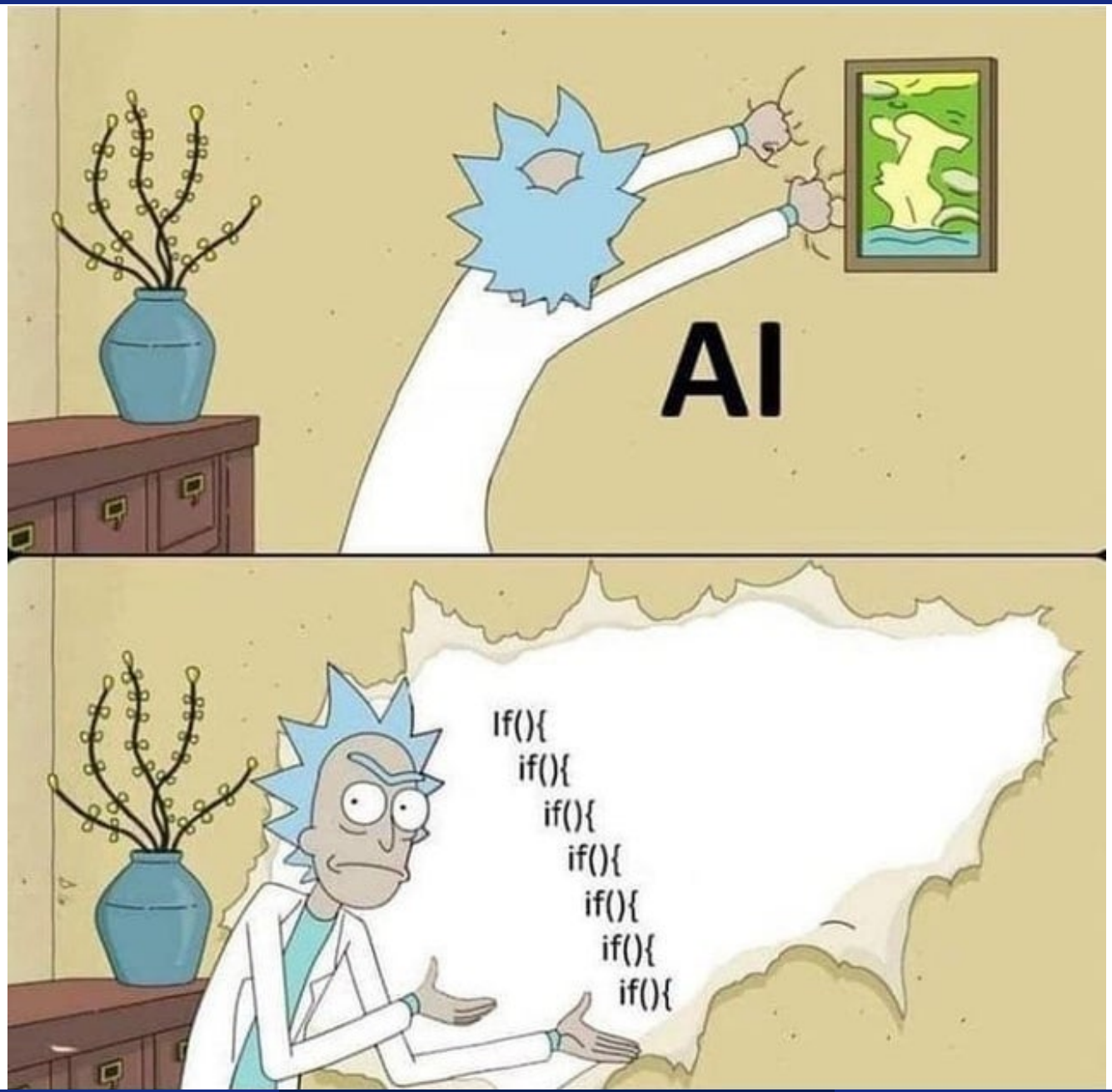
Conclusions - Interpretability ?

Discrete Transformer programs are easier to interpret than classic Transformers :

- Debuggin for certain problem instances becomes easier.
- Automatic analysis of the program can be done (e.g. formal verification).

But are *humanly* interpretable ? it depends...

Conclusions - Interpretability ? (cont.)



Conclusions - Interpretability ? (cont.)

```python
# First attention head: copy previous token.
def predicate_0_0(q_position, k_position):
    if q_position in {0, 13}:
        return k_position == 12
    elif q_position in {1}:
        return k_position == 0
    elif q_position in {2}:
        return k_position == 1
    elif q_position in {3}:
        return k_position == 2
    elif q_position in {4}:
        return k_position == 3
    elif q_position in {5}:
        return k_position == 4
    elif q_position in {6}:
        return k_position == 5
    elif q_position in {7}:
        return k_position == 6
    elif q_position in {8}:
        return k_position == 7
    elif q_position in {9}:
        return k_position == 8
    elif q_position in {10}:
        return k_position == 9
    elif q_position in {11}:
        return k_position == 10
    elif q_position in {12}:
        return k_position == 11
    elif q_position in {14}:
        return k_position == 13
    elif q_position in {15}:
        return k_position == 14
attn_0_0_pattern = select_closest(positions, positions,
                                  predicate_0_0)
attn_0_0_outputs = aggregate(attn_0_0_pattern, tokens)
```

```python
# MLP: reads current token and previous token
# Outputs 13 if it sees "()" or ")(".
def mlp_0_0(token, attn_0_0_output):
    key = (token, attn_0_0_output)
    if key in {("(", ")"),
               ("(", "}"),
               ("{", ")"),
               ("}", ")"),
               ("}", "}")}:
        return 4
    elif key in {(")", "{"),
                 ("}", "(")}:
        return 13
    elif key in {("(", ")"),
                 ("(", "}"),
                 (")", "("),
                 ("{", "}"),
                 ("}", "{")}:
        return 0
    return 7
mlp_0_0_outputs = [
    mlp_0_0(k0, k1) for k0, k1 in
    zip(tokens, attn_0_0_outputs)
]

# 2nd layer attention: check for "()" or ")("
def predicate_1_2(position, mlp_0_0_output):
    if position in {0, 1, 2, 4, 5, 6, 7, 8, 9,
                    10, 11, 12, 13, 14, 15}:
        return mlp_0_0_output == 13
    elif position in {3}:
        return mlp_0_0_output == 4
attn_1_2_pattern = select_closest(
    mlp_0_0_outputs, positions, predicate_1_2)
attn_1_2_outputs = aggregate(
    attn_1_2_pattern, mlp_0_0_outputs)
```

Conclusions - Interpretability ?

Obviously they are not written in the way a human would (especially for simple tasks), so they aren't very easy to understand.

On the other hand from a human perspective :

- Still we are able to follow the information flow (Simulatability, an aspect of transparency is sensibly improved with respect to classic Transformers).
- Such programs can be (manually) decomposed into a collection of high level interoperable functions.

References I

Plamen P. Angelov, Eduardo A. Soares, Richard Jiang, Nicholas I. Arnold, and Peter M. Atkinson.
Explainable artificial intelligence : an analytical review.
WIREs Data Mining and Knowledge Discovery, 11(5) :e1424, 2021.

Dan Friedman, Alexander Wettig, and Danqi Chen.
Learning transformer programs, 2023.

Prashant Gohel, Priyanka Singh, and Manoranjan Mohanty.
Explainable ai : current status and future directions, 2021.

David Lindner, János Kramár, Sebastian Farquhar, Matthew Rahtz, Thomas McGrath, and Vladimir Mikulik.
Tracr : Compiled transformers as a laboratory for interpretability, 2023.

Gail Weiss, Yoav Goldberg, and Eran Yahav.
Thinking like transformers.
In Marina Meila and Tong Zhang, editors, *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pages 11080–11090. PMLR, 18–24 Jul 2021.