

# Neural Decompiling of Tracr Transformers

Hannes Thurnherr<sup>[0009–0001–4554–3899]</sup> and Kaspar Riesen<sup>[0000–0002–9145–3157]</sup>

Institute of Computer Science, University of Bern, 3012 Bern, Switzerland  
`hannes.thurnherr@students.unibe.ch, kaspar.riesen@unibe.ch`

**Abstract.** Recently, the transformer architecture has enabled substantial progress in many areas of pattern recognition and machine learning. However, as with other neural network models, there is currently no general method available to explain their inner workings. The present paper represents a first step towards this direction. We utilize *Transformer Compiler for RASP* (Tracr) to generate a large dataset of pairs of transformer weights and corresponding RASP programs. Based on this dataset, we then build and train a model, with the aim of recovering the RASP code from the compiled model. We demonstrate that the simple form of Tracr compiled transformer weights is interpretable for such a decompiler model. In an empirical evaluation, our model achieves exact reproductions on more than 30% of the test objects, while the remaining 70% can generally be reproduced with only few errors. Additionally, more than 70% of the programs, produced by our model, are functionally equivalent to the ground truth, and therefore a valid decompilation of the Tracr compiled transformer weights.

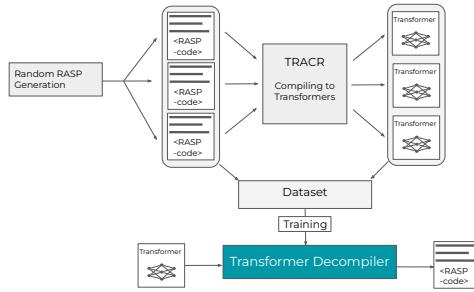
**Keywords:** Transformer · Interpretability · Decompiling · RASP

## 1 Introduction

The *transformer architecture* [1] is a type of neural network that was originally designed for machine translation but is now also successfully used for modelling many different pattern recognition tasks. This notably includes advanced language understanding [2] but also the modelling of non-sequential data such as images [3]. The transformer architecture differentiates itself from other architectures by using a mixture of self-attention and MLP-layers. It can be used as both a decoder-only variant or in an encoder-decoder configuration where the encoder augments the decoder forward pass using cross-attention.

*Interpretability* is a subfield of machine learning that is concerned with the problem of understanding the internals of neural networks [4]. Especially the interpretability of transformer-based models has seen growing attention in the past few years. While some progress has been made, e.g. [5,6,7], transformer neural networks are still largely regarded as black boxes. That is, interpretation relies on manual work which makes it hard to scale (for instance, manually translating billions of model weights and activations into human-readable descriptions is hardly feasible – even if one knew how to do so). The fact that current state-of-the-art systems are not interpretable, means that they could have unacceptable failure modes, which may only be discovered after deployment.

Attempts to automate at least parts of the complete interpretation pipeline have been made [8,9]. However, most of these approaches only focus on the analysis of a subcomponent of the network. In the present paper, we propose an end-to-end framework, where a decompiler model is responsible for the entire interpretation process of a complete neural network (taking all of its parameters into account). In particular, we investigate whether it is possible to train a deep-learning model to automate the translation of a set of compiled transformer weights, into *RASP*<sup>1</sup> code [10]. *RASP* is a programming language that consists of functions that correspond to various components of transformers and is designed to be computationally equivalent to transformers. *RASP* code could thus open up a way of translating transformation processes into a form that is more readable for humans.



**Fig. 1.** The core concept of the present method is based on the idea that Tracr can be used to generate training data for an automated end-to-end interpretability system.

One of the major challenges of the present research is the generation of a large amount of training data. To date, only a few examples of pairs of network weights and corresponding descriptions are available (from which a neural network could learn to translate between the two). To address this challenge, we use the *Transformer Compiler for Rasp* (Tracr) [11] to generate a novel set of training data. In particular, we use Tracr to produce a large set of pairs of transformer weights and corresponding *RASP* programs. We hypothesize that using these pairs to train a *Transformer Decompiler Model* could represent a first step towards a holistic and fully automated solution for transformer interpretability. In Fig. 1, the proposed approach of the present paper is visually summarized.

The remainder of the paper is organized as follows. Next, in Section 2, we describe in detail how the dataset, used to train our model, is engineered. Then, in Section 3, we outline the novel method for processing the *RASP* transformer weight pairs and training the decompiler model. In Section 4, we present and discuss the results of an experimental evaluation. Finally, in Section 5, we draw conclusions and discuss possible future research activities.

---

<sup>1</sup> *Restricted Access Sequence Processing Language*

## 2 Pairs of RASP Code and Transformer Weights

This section describes how we generate random but functional RASP code and how these RASP programs are then filtered and translated into transformer weights. This development of a novel dataset consisting of pairs of RASP code and the corresponding transformer weights is crucial for training the proposed model.

We use Algorithm 1 to create random, compilable RASP programs. This algorithm processes the primitives *rasp.tokens* and *rasp.indices* using the RASP functions *Select()*, *Aggregate()*, *SelectorWidth()*, *Map()* and *SequenceMap()* in a way that accounts for the compatibility of functions and the different datatypes. We achieve this by basing the algorithm around a pool of available inputs, which is, in turn, initialized containing only the two primitives *rasp.tokens* and *rasp.indices* (Line 2 of Algorithm 1)

Algorithm 1 starts by randomly selecting one of the five RASP functions to use next (Line 6). This random selection is weighted in accordance with how frequently the functions appear in a collection of manually written RASP programs. We then iterate through all of the parameters of the function (Line 7), randomly choosing a variable from the pool of available inputs and test whether it is of the datatype needed for the current parameter (Line 8 and 9). If we find that one or more of the parameters of the currently selected function cannot be satisfied by any of the currently available inputs, we choose a different function (Line 11). If this is the case for all of the functions, we restart the generation of the program (Line 12 and 13).

---

### Algorithm 1 Random RASP Program Generation

---

```

1: available_inputs ← Initialize with primitives rasp.tokens and rasp.indices
2: available_functions ← Select(), Aggregate(), SelectorWidth(), Map(), and SequenceMap()
3: available_lambdas ← 2-input lambdas, 1-input lambdas and 2-input, boolean output lambdas
4: phase ← 0
5: while not converged to one output do
6:   Select a function based on its probability from available_functions
7:   for each parameter of the function do
8:     if appropriate input available for the parameter then
9:       Use inputs from available_inputs and available_lambdas
10:    else
11:      Select a different function from available_functons
12:      if no function is compatible then
13:        Restart the program generation
14:      end if
15:    end if
16:   end for
17:   Add function's output to available_inputs
18:   if phase = 0 and termination criterion is met then
19:     phase ← 1
20:   end if
21:   if phase = 1 then
22:     Remove used inputs from available_inputs
23:   end if
24: end while
25: return The final set of operations forming the RASP program

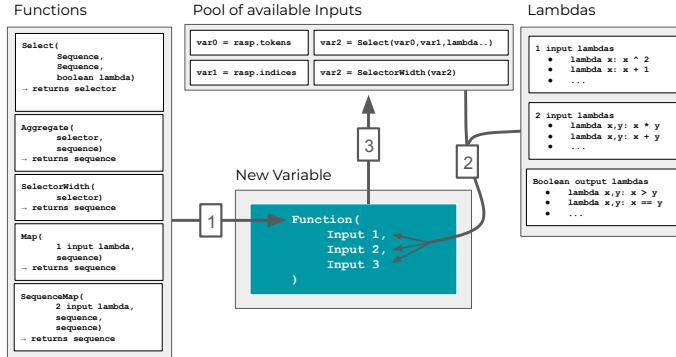
```

---

Once a suitable input for all of the function parameters has been found, the output of this function is represented by a new entry in the pool of available inputs (Line 17). Note that some functions, like *Map()* or *Select()*, take lambda functions as some of their inputs. These are not stored in the pool of available inputs but in external lists (Line 3).

In Algorithm 1, there are two phases (*phase 0* and *phase 1*). In *phase 0*, we expand the number of available inputs by leaving used variables in the pool of available inputs while adding new variables to this pool. The duration of *phase 0* is controlled via the termination criterion (Line 18). In our implementation, we check whether the length of (*available\_inputs* – 2) is greater than the duration of *phase 0*. Note that this criterion could be varied to guide the length of the generated programs. In *phase 1*, variables are removed from the available inputs once they are used. Since each function has only one output but typically multiple inputs, this process eventually converges to a single variable, representing the output of the program as a whole.

The output of Algorithm 1 is a computational graph represented as a list of nodes, each of which has a function and three inputs. Inputs, in turn, are represented by integers, denoting either the node they originate from, a lambda function, or an empty token for functions that take fewer than three inputs. Fig. 2 visually summarizes the process of Algorithm 1.



**Fig. 2.** Visualization of the three major steps of Algorithm 1 that are repeated until the pool of available inputs converges to one entry. 1: Select a function; 2: Fill the function with variables; 3: Add the newly created variable to the pool of available inputs.

As not all programs produced by Algorithm 1 are suitable for our purpose, we use a rejection sampling strategy. This means we test each program for undesired properties and only keep the ones that pass all the tests. To identify as many flaws or invalidities as possible, we run each program on a set of test inputs and employ the built-in Tracr validator. Properties that are filtered out relate to some limitations of RASP, like the inability to process float sequences in

**a: Error during compiling**

```

var0 = rasp.tokens
var1 = rasp.indices
var2 = SequenceMap(lambda x, y: x - y, var1, var1)
var3 = SequenceMap(lambda x,y:x/y if y!=0 else 0,var1, var0)
var4 = Select(var2, var2, lambda x, y: x > y)
var5 = Aggregate(var3, var4)

```

**b: More than two SequenceMap() uses:**

```

var0 = rasp.tokens
var1 = rasp.indices
var2 = Select(var0, var1, lambda x, y: x == y)
var3 = Select(var1, var0, lambda x, y: x < y)
var4 = SelectWidth(var2)
var5 = SequenceMap(lambda x, y: x + y, var4, var4)
var6 = SequenceMap(lambda x, y: (x + y) / 2, var5, var5)
var7 = SequenceMap(lambda x, y: x ** (1/y), var6, var3)
var8 = Aggregate(var7, var3)

```

**c: More than 15 lines**

```

var0 = rasp.tokens
var1 = rasp.indices
var2 = Select(var0, var0, lambda x, y: x > y)
var3 = Select(var0, var0, lambda x, y: x > y)
var4 = Select(var1, var1, lambda x, y: x > y)
var5 = SelectWidth(var2)
var6 = Select(var3, var5, lambda x, y: x == y)
var7 = Select(var4, var5, lambda x, y: x == y)
var8 = SelectWidth(var6)
var9 = Select(var7, var9, lambda x, y: x != y)
var10 = Select(var9, var9, lambda x, y: x != y)
var11 = SelectWidth(var3)
var12 = Select(var11, var11, lambda x, y: x > y)
var13 = SelectWidth(var10)
var14 = Map(lambda x: 1 if x > 0 else 0, var13)
var15 = Aggregate(var14, var8)
var16 = Aggregate(var15, var12)

```

**d: Validator found Issue: Categorical Aggregate with Selector with width > 1**

```

var0 = rasp.tokens
var1 = rasp.indices
var2 = Select(var0, var1, lambda x, y: x == y)
var3 = Select(var1, var0, lambda x, y: x < y)
var4 = SelectWidth(var2)
var5 = SequenceMap(lambda x, y: x + y, var4, var4)
var6 = SequenceMap(lambda x, y: (x + y)/2, var5, var5)
var7 = SequenceMap(lambda x, y: x** (1/y), var6, var3)
var8 = Aggregate(var7, var3)

```

**Fig. 3.** Four examples of generated programs that are rejected

the *Aggregate()* function and limitations of the Tracr compiler, which does not compile RASP code correctly if it involves the aggregation of non-categorical sequences using a selector with a width that is greater than one.

In Fig. 3, we show four examples of generated programs that are rejected due to different reasons:

- a:** The Tracr compiler can throw errors on certain valid RASP programs. Such programs are rejected.
- b:** Multiple uses of the function *SequenceMap()* can cause the compilation to take an unacceptably long time (several minutes rather than less than a second). We deal with this by filtering out programs with more than two occurrences of *SequenceMap()*.
- c:** The training set of programs is limited to 15 lines of code which amounts to a maximum of 60 program tokens.
- d:** The inbuilt Tracr validator filters programs where aggregation of non-categorical values using a selector wider than one occurs.

In total, we produce a dataset that consists of about 533,000 programs and corresponding transformers<sup>2</sup>. The produced programs contain between 5 and 12 lines of RASP code which equates to between 20 and 48 tokens. The corresponding transformers consist of between 8 and 42 weight matrices and therefore between 8 and 42 tokens.

To more closely examine the distribution of the programs, we run tests on a subset of 10,000 programs taken from the dataset. For instance, to determine the function<sup>3</sup> of each program, we generate a set of 1,000 input sequences. Programs which generate the same output to all of these inputs are then deemed to implement the same function.

<sup>2</sup> The generated data is publicly available upon request.

<sup>3</sup> By function we refer to the algorithm that is implemented by the RASP code.

During this evaluation, we observe that approximately 60% of the programs generate the same output for the 1,000 inputs as at least one other program in the dataset. However, the fact that certain functions are equal with respect to their input-output relationship does not necessarily mean that they are internally equal to each other. Actually, we observe that only about 5% of all programs refer to duplicates (i.e., pairs of programs with identical RASP code strings). However, it is worth mentioning that the small input space of the transformers that are compiled from the RASP programs could mean that two different RASP programs compile to the same transformer weights, leading to some ambiguity in the dataset, where multiple outputs (i.e. programs) would be correct for one input (i.e. weights).

### 3 The Transformer Decompiler Method (TraDe)

#### 3.1 Format of the Model Input

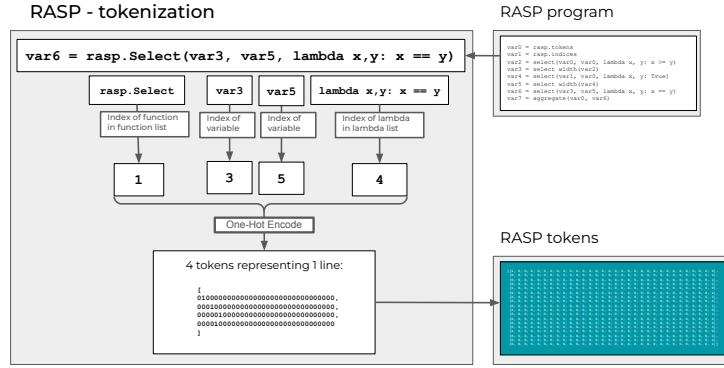
The formal representation of the RASP code (of the five component functions of the RASP code), into which the Tracr transformer weights are translated, is a crucial step in our procedure. It might be possible to use standard text tokenization [12] for this vectorization task. However, this would require the model to learn to distinguish valid RASP code from a very large space of possible outputs. Furthermore, since it is necessary to reverse the vectorization process, the application of commonly used methods for the vectorization of graphs, such as message passing [13], is not directly applicable to the computational graphs of the RASP program.

Based on these considerations, we employ a series of one-hot-encoded vectors (four per line of RASP code) to vectorize RASP code. Each series represents a specific part of the line, viz. the function and the three possible inputs to this function. Depending on the function, these numbers are interpreted differently. For instance, if the first vector of a line represents the function *SequenceMap()*, the next vector will be interpreted as the one-hot-encoded position of the lambda in the list of lambdas that produces one output from two inputs. In Fig. 4, the process of RASP vectorization is visualized

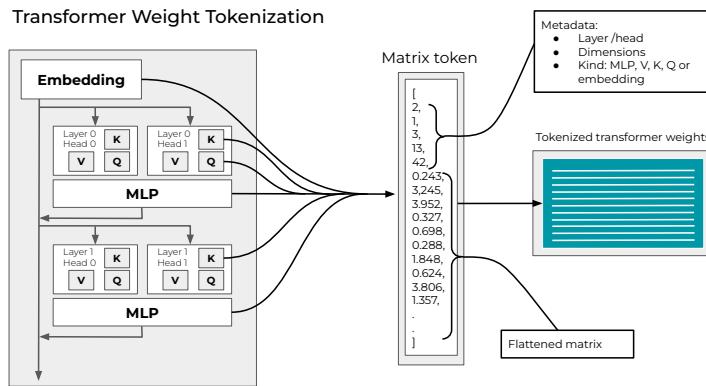
It is also necessary to bring the weights of the Tracr transformer into a form in which they can be fed into the decompiler. This task is not trivial as transformer weights are not organized sequentially. However, as demonstrated by vision transformers [3], the transformer architecture can also process sequences of tokens arranged sequentially, despite their original non-sequential configuration. To retain some of the structure of the transformer, we use a matrix-based tokenization<sup>4</sup>. That is, each token corresponds to a weight matrix in the compiled transformer. The matrix is flattened and concatenated with a small vector

---

<sup>4</sup> We also experiment with other tokenization schemes, such as layer-based tokenization or the naive flattening and partitioning into tokens, of all weights. Matrix-based tokenization emerged as superior in early experiments, though only by a small margin



**Fig. 4.** Illustration of the vector representation of the RASP code. It works by dividing each line into four components, which are in turn represented by a one-hot-encoded vector, denoting one of the options for this component.



**Fig. 5.** Illustration of the process of translating the weights of a compiled transformer into a set of tokens.

representing some information about the matrix such as shape, position in the transformer (head and layer) and function of the matrix (embedding, MLP, query, key or value). As the sizes of the matrices vary strongly, the resulting vectors are padded with zeros to achieve equal length. In Fig. 5, the process of transformer weight tokenization is visualized.

### 3.2 Decompiler Model

Our aim is to solve the problem of translating from the modality of transformer weights to the modality of RASP code. To this end, we adapt an existing system that also translates between modalities; namely the Whisper speech-to-text models by OpenAI [14]. Similar to Whisper, we employ an encoder-decoder transformer architecture. The encoder looks at the model input, in our case the tokenized transformer weights, and guides the decoder which produces the next RASP token based on all of the previously produced ones.

## 4 Experimental Evaluation

To evaluate our model, we generate an additional dataset of 1,000 programs, independent from the original 533K samples used for training. We optimize the hyperparameters of our model on one Nvidia GTX 3090. Note that the optimization of hyperparameters is based on the *token-accuracy* on a validation set of 28K samples. Token-accuracy refers to the fraction of output tokens that are correct relative to the total number of tokens. That is, every output token of our model is compared with the token at the corresponding position in the program that the transformer weights were compiled from. We are aware that the usefulness of this token-accuracy remains unclear for practical applications. However, it seems plausible that when a majority of the tokens are correct, this allows for the extraction of useful features from the RASP code, like for instance, the causal flow through the network.

The evaluated hyperparameters as well as the best-performing values for all hyperparameters are summarized in Table 4.

First, we evaluate the trained decompiler model on our independently generated test set in the non-autoregressive mode, in which it predicts the next token based on the ground-truth prefix.

In Fig. 6 we show the relative proportion of test programs that can be reproduced with a certain token-accuracy. We find that about 30% of all programs are reproduced identically to the ground truth (i.e., with a token accuracy of 100%)<sup>5</sup>. Approximately 85% of all test programs achieve a token-accuracy of 90% and overall we can report that all programs achieve at least 68% token-accuracy in this mode.

Moreover, in this mode we observe that 60% of the generated code is actually valid and runs without compilation error and 41% of the output programs are

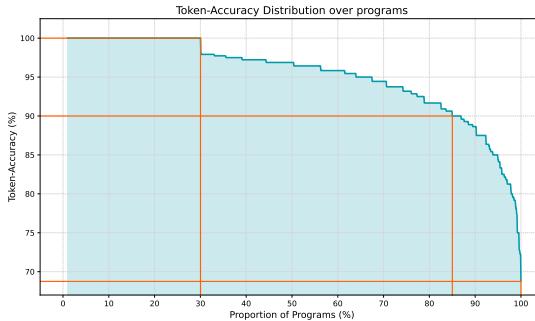
---

<sup>5</sup> We name the proportion of output programs that are identical to the ground truth *sequence equality*

Hyperparameter	Explored Range	Optimal Value
Feature Dimension	{512, 1024, 2048}	512
Heads per Layer (encoder & decoder)	{4,6,...,16}	16
Number of layers:		
Encoder Layers	{4,6,...16}	4
Decoder Layers	{4,6,...,16}	4
Feedforward Dimension	{1024, 2048}	2048
Dropout	{0.01, 0.1, 0.2, 0.3}	0.2
Input Dimension ( $x$ )		2000
Output Dimension ( $y$ )		32

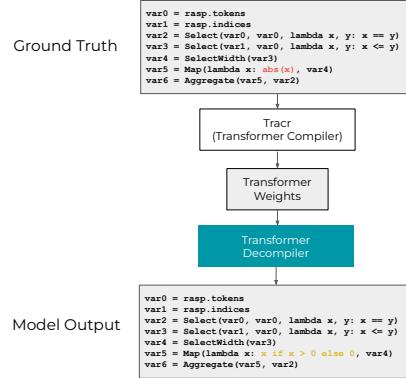
**Table 1.** Best performing values for all evaluated hyperparameters

functionally equivalent to the ground truth. This means that they represent the same input-output relations as the RASP code from which the transformer weights are originally compiled (an example of an output that does not match the ground truth but is functionally equivalent to the ground truth is shown in Fig. 7).

**Fig. 6.** Our model decompiles more than 30% of programs with 100% token-accuracy, with the rest of the programs being reproduced with at least 68% token-accuracy.

Next, we test the model in autoregressive mode, in which it produces each token based on the tokens it previously produced. In this mode, the relative number of programs that achieve 100% token accuracy (i.e., sequence equality) drops from 30% to 26%. However, 91% of the outputs are now compilable (rather than 60% as achieved in the non-autoregressive mode). Remarkably, 73% (rather than 41%) of the output programs are functionally equivalent to the ground truth RASP program that the transformer was compiled from. This could be explained by the model not seeing its previous outputs, but only the beginning of the ground-truth-program in the non-autoregressive mode. Seeing its own outputs for previous lines might allow the model to act according to the decisions it made earlier in the generation process.

The relative amount of outputs that achieve sequence equality, compilability, as well as functional equivalence are summarized in Fig. 8 for both modes (non-autoregressive and autoregressive).

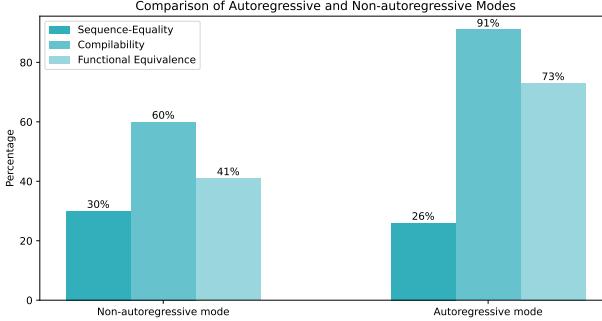


**Fig. 7.** These programs are identified as erroneous reproduction. However, they are functionally equivalent. The `var4` variable is produced by a `SelectWidth()` function, which only outputs values equal to or greater than zero. When applied to such values, the functions `abs(x)` and `x if x > 0 else 0` are equivalent.

## 5 Conclusion

To date, there are no general solutions available for the automatic interpretation of neural network models. In the present paper, we suggest that the Transformer Decompiler Method (TraDe) could serve as an approach to address this issue. In an empirical evaluation we show that TraDe enables a model to interpret the weights of other, smaller transformer neural networks and translate them into a more human-readable modality with useful accuracy. Our work thus represents a significant step towards an end-to-end framework for better interpretability.

However, there are still many limitations standing in the way of any practical application of the proposed concept. For instance, the transformer weights resulting from the Tracr compilation process are very different to transformer weights resulting from an optimization using stochastic gradient descent. The former is very sparse (except for certain structured elements), while the latter is very unstructured and dense. Moreover, the best-performing variant of the decompiler model is about three orders of magnitude larger than the models it is capable of decompiling (in terms of parameters). If this ratio is not significantly reduced, the application to modern, large transformer models will remain impossible. Lastly, though the step from weight matrices to RASP code is a large improvement in terms of interpretability, it would be wrong to call the



**Fig. 8.** Comparison of autoregressive and non-autoregressive mode across sequence equivalence to the ground truth, compilability and functional equivalence to the ground truth.

produced RASP programs human-readable. Even with the simple 5 to 12 lines of RASP code, it can take some minutes for a human to determine what sequence operation the algorithm implements.

Possible future work is concerned with reducing (or eliminating) the above-mentioned limitations. For instance, by taking the current decompiler model on a set of compressed Tracr models, it might be possible to adapt the decompiling Tracr transformer weights to the task of decompiling learned transformer weights (compressed Tracr models contain matrices that are trained with gradient descent, and thus might more closely reflect realistic weights). Another option might be to reduce the task of the decompiler from the reproduction of all RASP code to the detection of certain features, like backdoors [15] or deceptive tendencies [16,17], or to the analysis of a sub-component of the transformer [18]. Last but not least, it could also be interesting to see which weight matrix is attended to the most when producing a certain piece of RASP code.

**Acknowledgments.** This study was funded by the *Hasler Foundation Switzerland* (grant number 23085).

**Disclosure of Interests.** The authors have no competing interests to declare that are relevant to the content of this article.

## References

1. Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.
2. Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. 2018.

3. Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020.
4. Yu Zhang, Peter Tiňo, Aleš Leonardis, and Ke Tang. A survey on neural network interpretability. *IEEE Transactions on Emerging Topics in Computational Intelligence*, 5(5):726–742, 2021.
5. Trenton Bricken, Adly Templeton, Joshua Batson, Brian Chen, Adam Jermyn, Tom Conerly, Nick Turner, Cem Anil, Carson Denison, Amanda Askell, et al. Towards monosemanticity: Decomposing language models with dictionary learning. *Transformer Circuits Thread*, page 2, 2023.
6. Catherine Olsson, Nelson Elhage, Neel Nanda, Nicholas Joseph, Nova DasSarma, Tom Henighan, Ben Mann, Amanda Askell, Yuntao Bai, Anna Chen, et al. In-context learning and induction heads. *arXiv preprint arXiv:2209.11895*, 2022.
7. Steven Bills, Nick Cammarata, Dan Mossing, Henk Tillman, Leo Gao, Gabriel Goh, Ilya Sutskever, Jan Leike, Jeff Wu, and William Saunders. Language models can explain neurons in language models. *URL https://openaipublic.blob.core.windows.net/neuron-explainer/paper/index.html*. (Date accessed: 14.05.2023), 2023.
8. Ola Ahmad, Nicolas Béreux, Loïc Baret, Vahid Hashemi, and Freddy Lecue. Causal analysis for robust interpretability of neural networks. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, pages 4685–4694, 2024.
9. Arthur Conmy, Augustine Mavor-Parker, Aengus Lynch, Stefan Heimersheim, and Adrià Garriga-Alonso. Towards automated circuit discovery for mechanistic interpretability. *Advances in Neural Information Processing Systems*, 36, 2024.
10. Gail Weiss, Yoav Goldberg, and Eran Yahav. Thinking like transformers. In *International Conference on Machine Learning*, pages 11080–11090. PMLR, 2021.
11. David Lindner, János Kramár, Sebastian Farquhar, Matthew Rahtz, Tom McGrath, and Vladimir Mikulik. Tracr: Compiled transformers as a laboratory for interpretability. *Advances in Neural Information Processing Systems*, 36, 2024.
12. Lincoln A. Mullen, Kenneth Benoit, Os Keyes, Dmitry Selivanov, and Jeffrey Arnold. Fast, consistent tokenization of natural language text. *Journal of Open Source Software*, 3(23):655, 2018.
13. Clement Vignac, Andreas Loukas, and Pascal Frossard. Building powerful and equivariant graph neural networks with structural message-passing. *Advances in neural information processing systems*, 33:14143–14155, 2020.
14. Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine McLeavey, and Ilya Sutskever. Robust speech recognition via large-scale weak supervision. In *International Conference on Machine Learning*, pages 28492–28518. PMLR, 2023.
15. Xinyun Chen, Chang Liu, Bo Li, Kimberly Lu, and Dawn Song. Targeted backdoor attacks on deep learning systems using data poisoning. *arXiv preprint arXiv:1712.05526*, 2017.
16. Thilo Hagendorff. Deception abilities emerged in large language models. *arXiv preprint arXiv:2307.16513*, 2023.
17. Jérémie Scheurer, Mikita Balesni, and Marius Hobbahn. Large language models can strategically deceive their users when put under pressure. In *ICLR 2024 Workshop on Large Language Model (LLM) Agents*.
18. Lauro Langosco, William Baker, Neel Alex, David John Quarel, Herbie Bradley, and David Krueger. Towards meta-models for automated interpretability. “Manuscript in preparation”, 2024.