Tracr-Injection: Distilling Algorithms into Pre-trained Language Models

Tomás Vergara-Browne[*]   **Álvaro Soto**
Pontificia Universidad Católica de Chile
CENIA

tomas.vergarabrowne@mila.quebec

Abstract

Motivated by the surge of large language models, there has been a push to formally characterize the symbolic abilities intrinsic to the transformer architecture. A programming language, called RASP, has been proposed, which can be directly *compiled* into transformer weights to implement these algorithms. However, the tasks that can be implemented in RASP are often uncommon to learn from natural unsupervised data, showing a mismatch between theoretical capabilities of the transformer architecture, and the practical learnability of these capabilities from unsupervised data. We propose tracr-injection, a method that allows us to *distill* algorithms written in RASP directly into a pre-trained language model. We showcase our method by injecting 3 different algorithms into a language model. We show how our method creates an *interpretable subspace* within the model's residual stream, which can be decoded into the variables present in the code of the RASP algorithm. Additionally, we found that the proposed method can improve out-of-distribution performance compared to our baseline, indicating that indeed a more symbolic mechanism is taking place in the inner workings of the model. We release the code used to run our experiments[1].

1 Introduction

Transformer (Vaswani, 2017) based language models have become the main tool in NLP research. Recently, there has been growing interest in formally characterizing the types of mechanisms that this architecture can represent within its parameters. A key contribution in this research landscape is RASP (Weiss et al., 2021), a theoretical programming language whose main feature is its ability to map code instructions into weights of a transformer layers that implement the instructions. Additionally, tracr (Lindner et al., 2024) is a Python
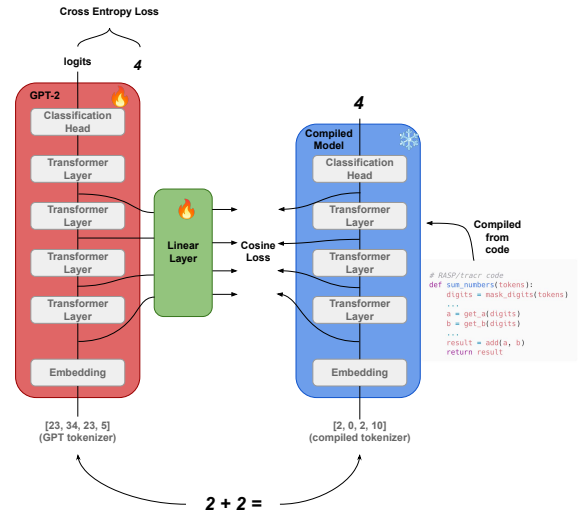

Figure 1: An overview of proposed method: we distill the residual stream of a compiled transformer using tracr, into a pre-trained model. We enforce the residual stream to *linearly encode* the compiled model's representations.

library that implements RASP compilation into actual transformer networks.

Many algorithms that are theoretically possible to implement in RASP (and therefore in a transformer), are not necessarily easy to learn for language models. An example is the recognition of *Dyck-n languages* (Ebrahimi et al., 2020), which involves recognizing the correct balance of n different parentheses within a string. For example, "([{}()])" correctly belongs to Dyck-3, while "([{}()])" does not. Some work has explored the difficulty of learning to recognize such languages in toy transformer networks (Ebrahimi et al., 2020), and pre-trained language models tend not to do well to recognize such data without undergoing extra finetuning (Zhang et al., 2024).

Previous work (Weng et al., 2023) improved the symbolic capabilities of language models by offloading token generation to a compiled RASP model in symbolic tasks. However, this approach relies on handcrafted rules to route token genera-

---

[*]Now at Mila and McGill University
[1]https://github.com/tvergara/tracr-injection

tion, raising questions about its advantage over simpler methods like directly executing external code. In contrast, we propose to *distill* algorithms implemented in RASP directly into the parameters of a pre-trained language model. Our method enhances the model's symbolic abilities *internally*, reducing inference complexity and directly boosting intrinsic symbolic capabilities. Moreover, our method provides direct access to *ground truth* variables underlying model predictions, significantly helping interpretability research, especially in automated circuit discovery (Conmy et al., 2023). These interpretability techniques aim to identify causally relevant components responsible for specific computations (e.g., the components responsible for computing additions) within language models. Evaluating these methods is challenging without ground truth references (Mosbach et al., 2024). Our method addresses this concern by explicitly injecting known algorithms as ground truth in pre-trained language models, enabling a novel way to benchmark and validate these interpretability methods. To the best of our knowledge, our work is the first to attempt to bridge this gap.

We summarize our contributions as follows:

- We introduce `tracr-injection`, a method to distill algorithms written in RASP into the inner workings of a pre-trained LLM.

- We demonstrate how to decode the residual stream (Elhage et al., 2021) of the final model into interpretable RASP variables, enhancing the interpretability of the mechanisms in the model.

- We show that the method uses the distilled representations from tracr models causally in their predictions, injecting a "ground truth" representation which the model causally uses to solve the task.

2 Related Work

RASP RASP, introduced by Weiss et al. (2021), is a theoretical programming language that maps code instructions directly to transformer weights. The main motivation of RASP was to provide a lower bound on the types of algorithms that are theoretically possible for transformer-based models to implement, and indeed some families of algorithms have been proven to be implementable inspired on this research (Yang and Chiang, 2024; Merrill and Sabharwal, 2023; Zhou et al., 2023). Building

on RASP, `tracr` (Lindner et al., 2024) provides a Python library to compile RASP algorithms into transformer weights. Recently, new alternatives to RASP and tracr have been proposed. ALTA (Shaw et al.) has also been proposed as an alternative programming language to RASP, with its own compiler publicly available. Smolensky et al. (2024) propose PSL, a Turing complete language which is also capable to be mapped to transformer weights.

Distillation Distillation is a well-established technique in machine learning (Hinton, 2015). Our approach uses a layer-wise distillation method—successfully applied in prior works (Romero et al., 2014; Jiao et al., 2019)—to inject tracr-compiled representations into a pre-trained LLM. To our knowledge, we are the first to distill compiled models using `tracr`. Perhaps the work most similar in spirit is Geiger et al. (2022), in which given a causal computational graph, they enforce machine learning models to encode the variables of this graph, and also train the model to follow the same causal relationships between the variables. They also test in o.o.d. settings to verify that their training procedure enforces a stronger generalization than a naive training. Although they do not categorize their method a distillation method, it is highly related.

Automatic Circuit Discovery In interpretability research, a machine learning model can be seen as a computational graph (Geiger et al., 2021), in which only a subgraph is responsible for a specific "mechanism" which might be human-understandable (Geiger et al., 2023). Automated Circuit Discovery techniques (Conmy et al., 2023), try to automatically compute the responsible subgraph for a mechanism, finding a minimal subset of nodes and edges to keep to explain the full behaviour of the model in some distribution of data. Research in this area has gained significant attention (Syed et al.; Marks et al., 2024; Ferrando et al., 2024; Bhaskar et al., 2024). However, the lack of a ground truth as to which is the "correct subgraph" is a significant limitation of this research area (Mosbach et al., 2024). Our method is related to this area in the sense that we try to address this concern by injecting a ground truth algorithm into the weights of a pre-trained model.

3 Methods

A diagram of our approach is shown in Figure 1. Our `tracr-injection` method for injecting an al-

gorithm into an LLM follows this procedure:

- We program and compile the desired algorithm using `tracr`. From this, we obtain a transformer model (and its tokenizer) which correctly implements the algorithm.

- We finetune a pre-trained LLM to *linearly encode* the residual stream of the compiled algorithm. This is done with a linear projection in the residual stream of the pre-trained model, and a cosine loss when comparing the projection to the residual stream of the compiled model. Figure 1 illustrates this process.

Specifically, let h^i_{model} be the residual stream activations (also known as *hidden states*) after the layer i in the model and let h^i_{mapped} be the same values but linearly mapped into a smaller dimension. Similarly, let h^i_{compiled} be these activations but in the model compiled with `tracr`. Let k be the number of layers of the compiled model. Our final loss is a sum of three components:

- L_{CE}: A cross-entropy loss on task data.

- L_{ALG}: An algorithm loss, aligning the pre-trained model's residual stream to the compiled model, defined as
  $\frac{1}{k} \sum_{i=0}^{k} 1 - \cos(h^i_{\text{mapped}}, h^i_{\text{compiled}})$

- L_{KL}: A KL-divergence loss on unsupervised text from Fineweb (Penedo et al., 2024).

The necessity of the L_{KL} loss arises from trying to maintain the model as close as possible as its pre-trained version, and avoid losing the general capabilities learned in pre-training. We balance the losses with hyperparameters α, β, γ (in practice we set $\alpha = \beta = \gamma = 1$).

$$L = \alpha L_{CE} + \beta L_{ALG} + \gamma L_{KL}. \quad (1)$$

To further enforce reliance on the learned symbolic subspace, we modify the final residual stream in k (after the last layer where there is an effect of the L_{ALG} loss) by projecting x onto the subspace defined by the linear layer ($h_{\text{proj}} = P_{\text{linear}} h^k_{\text{model}}$) and replacing its orthogonal component with the orthogonal component of another element in the batch:

$$h' = \texttt{roll}(h^k_{\text{model}} - h_{\text{proj}}) + h_{\text{proj}}, \quad (2)$$

where `roll` shifts activations along the batch dimension (using `detach` to prevent cross-batch gradients). This encourages the model to utilize the

injected symbolic representation subspace, without altering the activation distribution too much.

4 Algorithms

We implement in RASP algorithms for 3 tasks and compile them using `tracr` (with the `causal=True` option which compiles it into a causal transformer). These algorithms are:

- Recognition of Shuffle-Dyck: Shuffle-Dyck is a variation of dyck languages, in which the order between different parentheses is not a restriction, but only the balance of each one. For example, the string "[(])" does belong to Shuffle-Dyck. This language has been mostly studied in the RASP literature (Weiss et al., 2021), for its relatively easiness of implementation in RASP.

- Counting characters in a text: The transformer architecture is able to implement a specific counting logic (Yang and Chiang, 2024), however pre-trained language models have been shown on struggling to perform counting tasks (Yehudai et al., 2024). Therefore we add as a task the counting of "x" in a sequence of characters like "x x y x y y x y".

- Adding two numbers: Adding two numbers (as "312 + 89 =") in a causal transformer is non-trivial, and indeed many pre-trained LLMs are known to struggle with this sort of data (Qian et al., 2022).

5 Experiments

We distill each of these algorithms into GPT-2-large (Radford et al., 2019). We compare our approach with a strong baseline. This baseline consists of ablating the L_{ALG} loss from the loss function in equation 1, and removing our intervention described in equation 2. This baseline optimizes finetuning cross-entropy loss (L_{CE}) and KL loss on unsupervised text (L_{KL}), and hence we coin the baseline as Finetuning + KL. This baseline is expected to solve the tasks successfully and outperform our method in terms of distribution performance. This is due to the fact that the baseline does not contain the alternative objective L_{ALG} loss which does not directly optimize the in distribution performance. Additionally, it does not face the interventions described in Section 3 during training, which severely limit the types of mechanisms the model can potentially

learn during training to successfully solve the task. However, in the baseline, there are no guarantees regarding the mechanism the model is utilizing to solve the task, so there is no way to identify causal variables encoded in the model actually used.

To measure whether our method introduces catastrophic forgetting (Kirkpatrick et al., 2017), we evaluate the final models on tasks that probe both classification and language-modeling abilities:

Classification retention We run inference on the binary GLUE (Wang, 2018) tasks SST-2 (Socher et al., 2013) and MRPC (Dolan and Brockett, 2005), selecting the class with the highest logit.

Next-word prediction We measure accuracy on LAMBADA (Paperno et al., 2016), which tests long-range contextual understanding.

Language-model perplexity We compute perplexity on Tiny-Shakespeare (Blog, 2015), an unsupervised corpus disjoint from the finetuning data.

Only the perplexity measured on the o.o.d. Tiny-Shakespeare corpus appears in the main text (Table 1); the remaining results are deferred to Appendix (Table 3). All of our experiments share the same hyperparameters, which are shown in the Appendix B. We run each experiment with the same seed for reproducibility of our results.

6 Results

Our main results are shown in Table 1, where we only show the accuracy in the injected task, compared to the perplexity in a corpus of text not used during training (Blog, 2015). As we see, in all of our scenarios, the final accuracy in the test set shows a good performance, both for our baseline and our proposed method.

Also, we highlight that performance on all unrelated tasks remains high in the baseline, as evidenced by the perplexity scores. Additionally, the performance on 3 tasks is being maintained showed in our extended results in the Table 3 of the appendix. This serves as evidence that our method does not induce catastrophic forgetting while distilling an algorithm into the model.

7 Analysis

7.1 Symbolic Representations Learned

A key motivation for our work is to enforce symbolic reasoning in a pre-trained language model

Shuffle Dyck		
Method	**Accuracy ($\uparrow$)**	**Perplexity ($\downarrow$)**
No editing	0.0%	569.7
Finetuning + KL	100.0%	569.3
Tracr Injection	99.9%	569.9
Integer Addition		
Method	**Accuracy ($\uparrow$)**	**Perplexity ($\downarrow$)**
No editing	0.1%	569.7
Finetuning + KL	99.7%	570.1
Tracr Injection	95.0%	572.4
Count		
Method	**Accuracy ($\uparrow$)**	**Perplexity ($\downarrow$)**
No editing	0.0%	569.7
Finetuning + KL	99.8%	571.2
Tracr Injection	99.2%	571.4

Table 1: Performance in distilled task accuracy, and the perplexity in an o.o.d corpus of text using GPT-2 distilled by the proposed method, the baseline, and the original GPT-2 model. A full version of this table is in the Appendix, Table 3.

by injecting explicit RASP variables into its activations. We first verify that the model encodes these variables in its residual stream: our learned linear layer maps the residual stream activations to the compiled model's representation, enabling us to decode intermediate variables using their known labels (see Figure 2 and Appendix E).

To assess their causal role, we perturb the symbolic subspace by injecting noise. This degradation in performance is significantly larger than that caused by perturbing random directions, indicating that the injected symbolic representations are causally influential in the model's predictions (see Appendix A).

The encoding of ground truth variables that the model causally uses for its predictions is an important benefit of our proposed method. By being able to decode the residual stream activations into these interpretable variables, we are much closer to having a final model that we can understand *how* it solves the task. This is not the case with normal finetuning settings, including our baseline, as we have no strong methods to extract ground-truth representations which are causally relevant to solve the task at hand. The call for more transparent systems has been raised several times in recent years

Task	Test Data	TI	FT
Integer Sum	Cascading Overflow	47.8%	49.1%
Integer Sum	4 digit sums	0.9%	6.6%
Integer Sum	Decimals	0.0%	0.0%
Shuffle-Dyck	*almost-balanced* examples	96.6%	75.28%
Shuffle-Dyck	3x length	98.5%	88.6%
Shuffle-Dyck	Additional parentheses family	99.1%	84.3%
Count	X only data	21.9%	33.3%
Count	3x length	2.8%	3.5%
Count	Replace y → z	4.6%	3.0%

Table 2: Performance in out-of-distribution scenarios comparing Tracr-Injection (TI) and our finetuning + KL baseline (FT). A full description of each setting is in Appendix D.
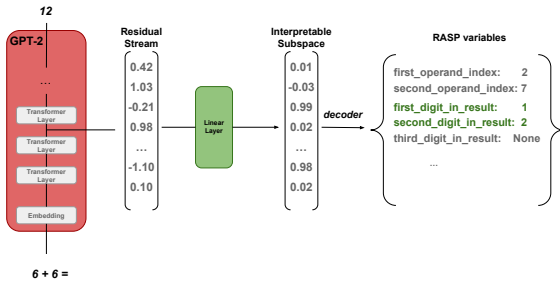

Figure 2: An example of how we can interpret the residual stream of the final model given the final linear layer used during training. Our decoder is given directly from our compiled model, where we know how each dimension in the residual stream maps to a particular value of a variable in the RASP program. An example of this process is found in Appendix E.

(Bengio et al., 2025; Casper et al., 2024), and our method is a step toward this goal.

7.2 Tracr-Injection can enhance o.o.d. generalization

A final test that would let us comfortably claim that we injected a ground truth algorithm into the model would be if the final model is able to make consistent predictions outside of the data it was initially trained on, showing that it is able to implement the RASP algorithm in its whole spectrum.

We design several out-of-distribution (o.o.d.) evaluations for each algorithm. Some of these tests are tasks that the compiled RASP model would solve perfectly, while others fall outside the scope of the compiled model due to the nature of the implemented algorithm. We expect Tracr-injection to be able to generalize to some extent in o.o.d. scenarios where the compiled transformer model supported the scenario, while not being able to generalize in other cases.

We show our results in Table 2. We find that our method can enhance the o.o.d. generalization in some settings. In the case of *shuffle-dyck*, the accuracy in all out-of-distribution settings practically did not suffer, while the baseline considerably achieved much lower scores. In other settings, the difference between our proposed method and the baseline is not as strong, perhaps suggesting that the overall o.o.d. ability is linked to the specific algorithm which is being distilled. We are optimistic that future work more focused on distilling the causal structure from the compiled network (Geiger et al., 2022) can have the potential to overcome these limitations and make o.o.d. capabilities generalize independent of the algorithm at hand.

8 Future Work

Our method provides the first step towards truly injecting a ground truth mechanism into a pre-trained language model. However, the lack of consistent o.o.d. generalization highlights that more work needs to be done in order to truly inject a mechanism that generalizes. One avenue of exploration is understanding why certain injected algorithms seem to generalize consistently better than other algorithms (as the shuffle-dyck algorithm). There might be connections between these results and RASP-L (Zhou et al., 2023). As RASP-L is a "learnable" subset of RASP instructions, it might be that some algorithms were inherently "harder" to truly learn than others. Additionally, scaling our method to more challenging tasks is a fruitful avenue forward, as all of our tasks could be easily learned by both the baseline and our method. Scaling to these more complex tasks comes with its own complications (as the implementation of ever more complex RASP programs), but it can highlight the benefits of using Tracr-Injection in comparison to finetuning.

Additionally, the possibility to inject a ground truth mechanism into pre-trained LLMs is exciting for interpretability research. A big problem in interpretability research is the lack of "ground truth". This is a limitation when evaluating methods that, for example, automatically extract a "circuit" for a task (Conmy et al., 2023). This lack of "ground truth" is in fact a common criticism in interpretability research in NLP (Mosbach et al., 2024). With the possibility of injecting a "ground truth" to LLMs inner workings, a door is opened to start benchmarking interpretability methods based on injected ground truths.

9 Limitations

While our methods do provide a way of distilling the RASP variables into the residual stream of a transformer, the out-of-distribution tests suggests the model is not representing these same variables when tested in examples which are radically outside of the distribution seen in training. This is a big limitation, as a fully injected algorithm should work in all distributions of inputs, and not just in a vicinity. Additionally, the use of 3 simple symbolic tasks to showcase our method is another limitation. The injection of more complex RASP programs can potentially highlight the performance benefits of our method in constrast to finetuning. The final limitation we mention is the focus exclusively on GPT-2 to evaluate the performance of our work. Compute is an important restriction for which we have made this decision, and we encourage further work in scaling up our methods into more (and larger) language models.

10 Acknowledgements

References

Yoshua Bengio, Sören Mindermann, Daniel Privitera, Tamay Besiroglu, Rishi Bommasani, Stephen Casper, Yejin Choi, Philip Fox, Ben Garfinkel, Danielle Goldfarb, et al. 2025. International ai safety report. *arXiv preprint arXiv:2501.17805*.

Adithya Bhaskar, Alexander Wettig, Dan Friedman, and Danqi Chen. 2024. Finding transformer circuits with edge pruning. *Advances in Neural Information Processing Systems*, 37:18506–18534.

Andrej Karpathy Blog. 2015. The unreasonable effectiveness of recurrent neural networks. *URL: http://karpathy. github. io/2015/05/21/rnn-effectiveness/dated May*, 21:31.

Stephen Casper, Carson Ezell, Charlotte Siegmann, Noam Kolt, Taylor Lynn Curtis, Benjamin Bucknall, Andreas Haupt, Kevin Wei, Jérémy Scheurer, Marius Hobbhahn, et al. 2024. Black-box access is insufficient for rigorous ai audits. In *Proceedings of the 2024 ACM Conference on Fairness, Accountability, and Transparency*, pages 2254–2272.

Arthur Conmy, Augustine Mavor-Parker, Aengus Lynch, Stefan Heimersheim, and Adrià Garriga-Alonso. 2023. Towards automated circuit discovery for mechanistic interpretability. *Advances in Neural Information Processing Systems*, 36:16318–16352.

Bill Dolan and Chris Brockett. 2005. Automatically constructing a corpus of sentential paraphrases. In *Third international workshop on paraphrasing (IWP2005)*.

Javid Ebrahimi, Dhruv Gelda, and Wei Zhang. 2020. How can self-attention networks recognize dyck-n languages? *arXiv preprint arXiv:2010.04303*.

Nelson Elhage, Neel Nanda, Catherine Olsson, Tom Henighan, Nicholas Joseph, Ben Mann, Amanda Askell, Yuntao Bai, Anna Chen, Tom Conerly, Nova DasSarma, Dawn Drain, Deep Ganguli, Zac Hatfield-Dodds, Danny Hernandez, Andy Jones, Jackson Kernion, Liane Lovitt, Kamal Ndousse, Dario Amodei, Tom Brown, Jack Clark, Jared Kaplan, Sam McCandlish, and Chris Olah. 2021. A mathematical framework for transformer circuits. *Transformer Circuits Thread*. Https://transformer-circuits.pub/2021/framework/index.html.

Javier Ferrando, Gabriele Sarti, Arianna Bisazza, and Marta R Costa-jussà. 2024. A primer on the inner workings of transformer-based language models. *arXiv preprint arXiv:2405.00208*.

Atticus Geiger, Duligur Ibeling, Amir Zur, Maheep Chaudhary, Sonakshi Chauhan, Jing Huang, Aryaman Arora, Zhengxuan Wu, Noah Goodman, Christopher Potts, et al. 2023. Causal abstraction: A theoretical foundation for mechanistic interpretability. *arXiv preprint arXiv:2301.04709*.

Atticus Geiger, Hanson Lu, Thomas Icard, and Christopher Potts. 2021. Causal abstractions of neural networks. *Advances in Neural Information Processing Systems*, 34:9574–9586.

Atticus Geiger, Zhengxuan Wu, Hanson Lu, Josh Rozner, Elisa Kreiss, Thomas Icard, Noah Goodman, and Christopher Potts. 2022. Inducing causal structure for interpretable neural networks. In *International Conference on Machine Learning*, pages 7324–7338. PMLR.

Geoffrey Hinton. 2015. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*.

Xiaoqi Jiao, Yichun Yin, Lifeng Shang, Xin Jiang, Xiao Chen, Linlin Li, Fang Wang, and Qun Liu. 2019. Tinybert: Distilling bert for natural language understanding. *arXiv preprint arXiv:1909.10351*.

James Kirkpatrick, Razvan Pascanu, Neil Rabinowitz, Joel Veness, Guillaume Desjardins, Andrei A Rusu, Kieran Milan, John Quan, Tiago Ramalho, Agnieszka Grabska-Barwinska, et al. 2017. Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences*, 114(13):3521–3526.

David Lindner, János Kramár, Sebastian Farquhar, Matthew Rahtz, Tom McGrath, and Vladimir Mikulik. 2024. Tracr: Compiled transformers as a laboratory for interpretability. *Advances in Neural Information Processing Systems*, 36.

Samuel Marks, Can Rager, Eric J Michaud, Yonatan Belinkov, David Bau, and Aaron Mueller. 2024. Sparse feature circuits: Discovering and editing interpretable causal graphs in language models. *arXiv preprint arXiv:2403.19647*.

William Merrill and Ashish Sabharwal. 2023. A logic for expressing log-precision transformers. *Advances in neural information processing systems*, 36:52453–52463.

Marius Mosbach, Vagrant Gautam, Tomás Vergara-Browne, Dietrich Klakow, and Mor Geva. 2024. From insights to actions: The impact of interpretability and analysis research on nlp. *arXiv preprint arXiv:2406.12618*.

Denis Paperno, Germán Kruszewski, Angeliki Lazaridou, Quan Ngoc Pham, Raffaella Bernardi, Sandro Pezzelle, Marco Baroni, Gemma Boleda, and Raquel Fernández. 2016. The lambada dataset: Word prediction requiring a broad discourse context. *arXiv preprint arXiv:1606.06031*.

Guilherme Penedo, Hynek Kydlíček, Anton Lozhkov, Margaret Mitchell, Colin Raffel, Leandro Von Werra, Thomas Wolf, et al. 2024. The fineweb datasets: Decanting the web for the finest text data at scale. *arXiv preprint arXiv:2406.17557*.

Jing Qian, Hong Wang, Zekun Li, Shiyang Li, and Xifeng Yan. 2022. Limitations of language models in arithmetic and symbolic induction. *arXiv preprint arXiv:2208.05051*.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. 2019. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9.

Adriana Romero, Nicolas Ballas, Samira Ebrahimi Kahou, Antoine Chassang, Carlo Gatta, and Yoshua Bengio. 2014. Fitnets: Hints for thin deep nets. *arXiv preprint arXiv:1412.6550*.

Peter Shaw, James Cohan, Jacob Eisenstein, Kenton Lee, Jonathan Berant, and Kristina Toutanova. Alta: Compiler-based analysis of transformers. In *The First Workshop on System-2 Reasoning at Scale, NeurIPS'24*.

Paul Smolensky, Roland Fernandez, Zhenghao Herbert Zhou, Mattia Opper, and Jianfeng Gao. 2024. Mechanisms of symbol processing for in-context learning in transformer networks. *arXiv preprint arXiv:2410.17498*.

Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D Manning, Andrew Y Ng, and Christopher Potts. 2013. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 conference on empirical methods in natural language processing*, pages 1631–1642.

Aaquib Syed, Can Rager, and Arthur Conmy. Attribution patching outperforms automated circuit discovery. In *The 7th BlackboxNLP Workshop*.

A Vaswani. 2017. Attention is all you need. *Advances in Neural Information Processing Systems*.

Alex Wang. 2018. Glue: A multi-task benchmark and analysis platform for natural language understanding. *arXiv preprint arXiv:1804.07461*.

Gail Weiss, Yoav Goldberg, and Eran Yahav. 2021. Thinking like transformers. In *International Conference on Machine Learning*, pages 11080–11090. PMLR.

Yixuan Weng, Minjun Zhu, Fei Xia, Bin Li, Shizhu He, Kang Liu, and Jun Zhao. 2023. Mastering symbolic operations: Augmenting language models with compiled neural networks. *arXiv preprint arXiv:2304.01665*.

Andy Yang and David Chiang. 2024. Counting like transformers: Compiling temporal counting logic into softmax transformers. *arXiv preprint arXiv:2404.04393*.

Gilad Yehudai, Haim Kaplan, Asma Ghandeharioun, Mor Geva, and Amir Globerson. 2024. When can transformers count to n? *arXiv preprint arXiv:2407.15160*.

Zhehao Zhang, Jiaao Chen, and Diyi Yang. 2024. Darg: Dynamic evaluation of large language models via adaptive reasoning graph. *arXiv preprint arXiv:2406.17271*.

Hattie Zhou, Arwen Bradley, Etai Littwin, Noam Razin, Omid Saremi, Josh Susskind, Samy Bengio, and Preetum Nakkiran. 2023. What algorithms can transformers learn? a study in length generalization. *arXiv preprint arXiv:2310.16028*.

A Learned variables are causally influential

Although our motivation is to make the models prediction to function symbolically, and we have shown that the model effectively encode the symbolic representations that we want, these encodings might not be *causally* involved in the final prediction of the model. To verify that the models use these encodings causally, we design an experiment where we add noise in the residual stream of the model at the last layer which the pre-trained model had an algorithm loss (in which it should already have all the symbolic variables encoded). We compare two ways of adding noise:

- Adding noise on the direction of were the answer is encoded in the linear layer.

Shuffle Dyck					
Method	**Accuracy (↑)**	**Perplexity (↓)**	**SST-2 (↑)**	**MRPC (↑)**	**LAMBADA (↑)**
No editing	0.0%	569.7	67.4%	63.7%	58.8%
Finetuning + KL	100.0%	569.3	65.1%	58.8%	58.8%
Tracr Injection	99.9%	569.9	75.2%	66.7%	58.8%
Integer Addition					
Method	**Accuracy (↑)**	**Perplexity (↓)**	**SST-2 (↑)**	**MRPC (↑)**	**LAMBADA (↑)**
No editing	0.1%	569.7	67.4%	63.7%	58.8%
Finetuning + KL	99.7%	570.1	67.2%	66.2%	58.7%
Tracr Injection	95.0%	572.4	65.0%	65.9%	58.9%
Count					
Method	**Accuracy (↑)**	**Perplexity (↓)**	**SST-2 (↑)**	**MRPC (↑)**	**LAMBADA (↑)**
No editing	0.0%	569.7	67.4%	63.7%	58.8%
Finetuning + KL	99.8%	571.2	71.0%	61.8%	58.7%
Tracr Injection	99.2%	571.4	64.0%	65.7%	58.8%

Table 3: Performance in distilled task accuracy, and unrelated tasks using GPT-2 distilled by the proposed method, the baseline, and the original GPT-2 model.

- Adding noise in random directions (but the same number of dimensions as the previous subspace).

Our results for this experiment on *shuffle-dyck* is shown in Figure 3. We see that for any magnitude of noise, adding noise in the directions of were the answer is encoded in the linear layer is more hurtful towards the loss, than in random directions. This suggests partially that the model is using this direction causally in its predictions, more than random directions.

B Hyperparameters

For each experiment we use the exact same hyperparameter settings:

- 80k different training batches of synthetic data on the task.

- Learning rate of 10^{-6} in GPT-2, and 10^{-4} in the linear layer.

- Batch size of 12 task examples and 12 examples of unsupervised text from Penedo et al. (2024).

- Number of tokens per batch example of unsupervised data of 70.

- The seed used was 42.

Each training run was made on a single A40 48 GB GPU for approximately 10 hours.

C Details on each implemented task

We explain an example of each implemented task below:

- **Count task**
  - Example: "What is the number of 'X' in this text? x y x x y. Answer: 3"
  - We only do supervision on the final answer tokens (the 3 in this case). We generate a random selection of all the possibilities of cases of this task up until 40 total "x" or "y" characters, such that no example is repeated in training.

- **Integer sum task**
  - Example: "211 + 123 = 334"
  - We only do supervision on the final answer tokens (the 334 in this case). We generate all the combinations of sums of values until 450, which implies that the final result is not more than 3 digits (our compiled model with RASP manages up until 3 digit sums). Again, no example is repeated during training.
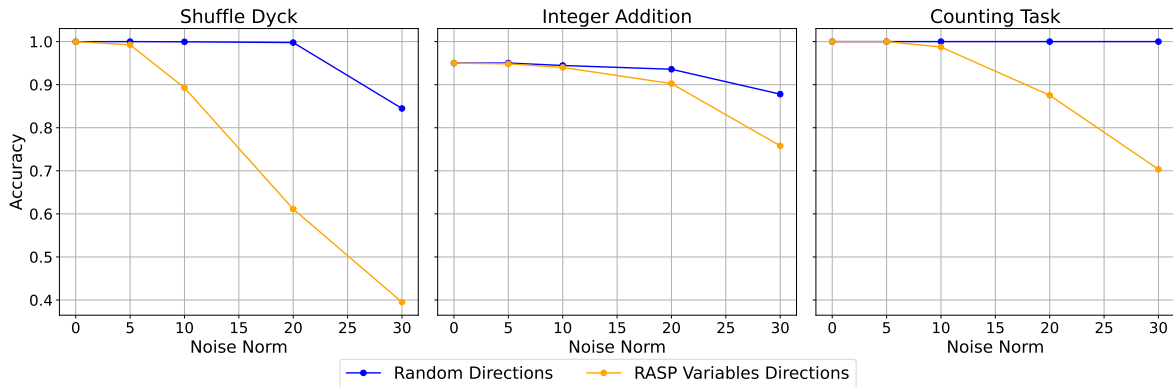
- **Shuffle Dyck task**

Figure 3: Accuracy in a model distilled with our algorithms. We add noise of different norms in the residual stream in the subspace where the final RASP variables are being encoded encode, and compare it to adding noise in random directions. A considerably larger effect is being made by the noise in the RASP variables subspace.

- Example: "Are parentheses here correctly matched? []{(}).Answer: Yes"

- We only do supervision on the final answer tokens (the 'yes' token in this case). This example is a correct one because in *shuffle dyck* the order between different parentheses families is not a restriction (so it does not matter that we close the "}" before closing the)). We generate random examples from which half are correct, and the other half are incorrect with sizes up to 30 parentheses in total. The parentheses pairs that we use are: "()", "{}" and "[]". Again, no example is repeated during training.

Additionally, in number related tasks (count and integer sum), we had to make the compiled tokenizer compatible with GPT-2 into how does it separate the training data. This implied removing multidigit tokens from the tokenizer (for example, if the string "123" was getting tokenized as only one token, we remove the token from tokenizer's vocabulary such that it would get tokenized in 3 different tokens, one for each digit).

D Out-of-distribution examples

Below we describe the examples we used in each of our evaluations on out-of-distribution settings. We chose 3 settings for each task, with one of these settings being a "tracr-supported" setting, which means that the original tracr-compiled model from the RASP code would solve this setting perfectly.

- **Count task:**

- X-only data: we use 40 examples with only "x" values, so for example "x x x x x", from 1 to 40 "x"s (as it was trained to only see sequences up to 40). It has to count every single character. These examples were not seen in training. The compiled model can solve this scenario.

- 3x length: we draw examples from 41 up to 120 characters long (3x with respect to the data it was trained on). It is not supported by the tracr model as it was compiled to manage sequences up to 60 of context length.

- Replace 'y' with 'z': We use the original test set, but replacing the 'y' characters with 'z'. This should be an invariant in the model. The compiled model would be able to solve this, as these tokens do not affect its computation.

- **Integer Sum task:**

- Cascading Overflow: We test it on data which the model has to overflow digits all the way from the right to the left, for example 299 + 1 (the model has to push the overflow digit all the way to the beginning). It is a main difficulty in implementing a sum algorithm in RASP, so we expected our method to work better. The compiled model can solve this scenario.

- 4 digit sums: We test it in data where each input is 4 digits long, and in training the inputs where at most 3 digits long. Our compiled model does not solve this scenario, as it was compiled to manage

inputs up to 3 digits.

– Decimal sums: We test it in data where we append a decimal part to each value to sum. Our compiled model does not solve this scenario, as it requires an extension of the algorithm for this.

- **Shuffle Dyck task:**

  – Almost-balanced examples: We create correct examples, and take away the last parentheses of the example, making it unbalanced. This evaluation tests the real decision boundaries to recognize that an example is balanced or not. Our compiled model supports this scenario.

  – 3x length: We test in data which is up to 90 parentheses long (3x with respect to the training data). Our compiled algorithm does not support this scenario, as it can handle inputs up to 90 tokens.

  – Additional parentheses family: We add the parentheses family "<>" which is unseen in the training data. Our compiled model does not solve this scenario, as only 3 parentheses families where specified in the compilation ("()", "{}", "[]").

E Walkthrough the decoding process

While a high level explanation of the decoding process can be found in Figure 2, we leave a detailed example of this decoding process using the use case of the *shuffle-dyck* task, with an input.

First, the RASP algorithm to solve this task can be found in Algorithm 1. For a full understanding of the algorithm, we strongly suggest reading the RASP and tracr papers (Weiss et al., 2021; Lindner et al., 2024), as the syntax and instructions specific to RASP will become clearer. However, for the purpose of our contribution, its only important to notice the **bold variables**. Each of these variables are encoded in one-hot in a subset of the neurons in the residual stream of the compiled model. For a single variable, it has a specific value according to the token position, so in order to decode a variable we also need to set fixed the position it is being decoded at. An example is the default variable `indices`, which is just the positional encoding. In an arbitrary selection of tokens like $[12, 25, 74]$", decoding the `indices` variable would give us $[0, 1, 2]$". Using the example of the algorithm, the variable `left_start_counts` gives
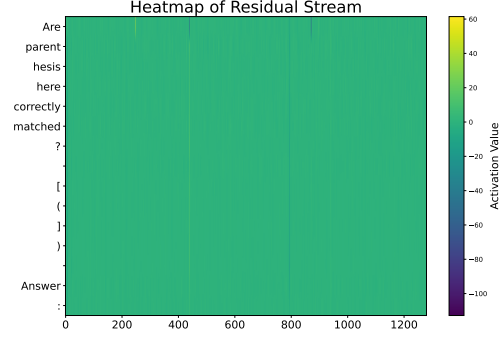

Figure 4: A plot of the residual stream in every token of the string "Are parentheses here correctly matched? [(]). Answer:" in a GPT-2 with the *shuffle-dyck* distiled on it. Interpreting these activations is impossible to the naked eye.

us the number of "starts" seen. For example the input "(())()" would give this variable the value $[1, 2, 2, 2, 3, 3]$".

So the process of decoding the residual stream from the trained GPT-2's residual stream becomes:

- Translating the residual stream activations into the compiled model's residual stream space by using the linear layer trained.

- For each token, and for each variable, we `argmax` the neurons corresponding to them, to get a value.

Let us take the example of how the a GPT-2 distilled with this algorithm can correctly predict the next token in a string like: "Are parentheses here correctly matched? [(]). Answer:". A heatmap of the activations of the residual stream of the model is seen in Figure 4. We see that the activations in each token are completely uninterpretable, however, when we pass these activations through the linear layer learned during distillation, we get Figure 5. These activations are much more sparse, and we can even see clear phenomena, as straight lines. Straight lines are values which are constant over all the tokens (many variables are indeed shared independent of the token position).

Finally we can map these activations into a table of the shape (`tokens, variables`) as in Figure 6, where we decode the information written for each variable in each token position. This process just implies taking the argmax over the neurons encoding all the values in a variable in the residual stream, for each of the tokens.
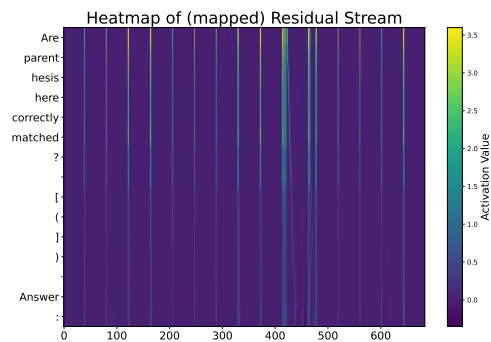
Figure 5: A plot of the residual stream in the token of the string "Are parentheses here correctly matched? [(]). Answer:", translated to a sparser space with the linear layer learned during distillation. Each straight line represents a value that is constant in every token, while other variables (like the inclined line) represent variables which change value for each token dimension.

F Prompts used for classification evaluation

Prompt segment 2: MRPC Prompt

```
Determine whether the following
two sentences are paraphrases (i.e
., have the same meaning). Answer
in one word: yes or no.

Sentence 1: The company reported
strong earnings this quarter.
Sentence 2: The business announced
 impressive profits for the
quarter.
Answer: yes

Sentence 1: I love to read science
 fiction novels.
Sentence 2: I prefer watching
documentaries over reading books.
Answer: no

Sentence 1: {sentence1}
Sentence 2: {sentence2}
Answer:
```

Prompt segment 1: SST2 Prompt

```
What is the sentiment in this
review? The answer should be one
word: either positive or negative.

Review: the movie was absolutely
incredible.

Answer: positive

What is the sentiment in this
review? The answer should be one
word: either positive or negative.

Review: this was a shameful waste
of time.

Answer: negative

What is the sentiment in this
review? The answer should be one
word: either positive or negative.

Review: {sentence}

Answer:
```

	Are	parent	hesis	here	correctly	matched	?		[	(	]	)		Answer	:
()_diffs_39	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0
()_negative_counters_36	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
()_negative_counters_36_selector_width_attn_output	True	True	True	True	True	True	True	True	True	True	True	True	True	True	True
(_start_counts_45	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1
(_start_counts_45_selector_width_attn_output	True	True	True	True	True	True	True	True	True	True	True	True	True	True	True
)_end_counts_46	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1
)_end_counts_46_selector_width_attn_output	True	True	True	True	True	True	True	True	True	True	True	True	True	True	True
[]_diffs_35	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0
[]_negative_counters_33	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
[]_negative_counters_33_selector_width_attn_output	True	True	True	True	True	True	True	True	True	True	True	True	True	True	True
[_start_counts_41	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1
[_start_counts_41_selector_width_attn_output	True	True	True	True	True	True	True	True	True	True	True	True	True	True	True
]_end_counts_42	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1
]_end_counts_42_selector_width_attn_output	True	True	True	True	True	True	True	True	True	True	True	True	True	True	True
aggregated_diffs_31	0	0	0	0	0	0	0	0	1	1	1	0	0	0	0
aggregated_diffs_tmp_0_34	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0
aggregated_negatives_30	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
final_result_28	False	False	False	False	False	False	False	False	False	False	False	False	False	False	True
indices	0	1	2	3	4	5	6	7	8	9	10	11	12	13	16
one	True	True	True	True	True	True	True	True	True	True	True	True	True	True	True
sequence_map_32	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
tokens									[	(	]	)			compute
unfiltered_result_29	True	True	True	True	True	True	True	True	False	False	False	True	True	True	True
{_start_counts_47	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
{_start_counts_47_selector_width_attn_output	True	True	True	True	True	True	True	True	True	True	True	True	True	True	True
{}_diffs_40	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
{}_negative_counters_37	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
{}_negative_counters_37_selector_width_attn_output	True	True	True	True	True	True	True	True	True	True	True	True	True	True	True
}_end_counts_48	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
}_end_counts_48_selector_width_attn_output	True	True	True	True	True	True	True	True	True	True	True	True	True	True	True

Figure 6: Decoded variables in the residual stream of a GPT-2 distilled with the Algorithm 1 when passed through the input "Are parentheses here correctly matched? [()]. Answer:".

Algorithm 1: Shuffle-Dyck RASP code

Function shuffle_dyck(pairs = ['()',
'{}', '[]']):
    all_diffs ← [], all_negs ← []
    foreach *(left, right) in pairs* **do**
        starts ← SELECT(tokens, token ==
         left)
        left_start_counts ←
         COUNT(starts)
        ends ← SELECT(tokens, token ==
         right)
        right_end_counts ←
         COUNT(ends)
        // Compute diffs
        left_right_diffs ←
         left_start_counts −
         right_end_counts
        Append **left_right_diffs** to all_diffs
        // Count negative differences
        negs_selector ← SELECT(
         left_right_diffs, value < 0)
        left_right_negative_counters ←
         COUNT(negs_selector)
        Append
         left_right_negative_counters to
         all_negs
    end
    // Aggregate negative counters
    current_negs ← all_negs[0]
    foreach *negs in all_negs[1:]* **do**
        current_negs ← MAP((x,y) → if (x
         $\neq 0$ or $y \neq 0$) then 1 else 0,
         current_negs, negs)
    end
    aggregated_negatives ← current_negs
    // Aggregate all balances
    current_diffs ← all_diffs[0]
    foreach *diffs in all_diffs[1:]* **do**
        current_diffs ← MAP((x,y) → if (x
         $\neq 0$ or $y \neq 0$) then 1 else 0,
         current_diffs, diffs)
    end
    unfiltered_result ← MAP((x,y) → (x
     and y == 0), aggregated_negatives,
     current_diffs)
    final_result ← MAP((x, token) → if
     (token == "compute") then x else False,
     unfiltered_result, tokens)
    return final_result