Journal of Artificial Intelligence Research 84 (2025) Article 23 Submitted 10/24; published 11/25

Mechanisms of Symbol Processing for In-Context Learning in Transformer Networks Online Appendices

Paul Smolensky psmo@microsoft.com, smolensky@jhu.edu
Microsoft Research Deep Learning, Redmond WA 98052, USA
Johns Hopkins University Cognitive Science, Baltimore MD 21218, USA

Roland Fernandez rfernand@microsoft.com
Microsoft Research Deep Learning, Redmond WA 98052, USA

Zhenghao Herbert Zhou herbert.zhou@yale.edu
Yale University Linguistics, New Haven CT 06511, USA

Mattia Opper M.Opper@ed.ac.uk
University of Edinburgh ILCC, Edinburgh EH8 9AB, UK

Adam Davies adavies4@illinois.edu
University of Illinois School of Computing, Urbana IL 61801, USA

Jianfeng Gao jfgao@microsoft.com
Microsoft Research Deep Learning, Redmond WA 98052, USA

©2025 The Authors. Published by AI Access Foundation under Creative Commons Attribution License CC BY 4.0.

Contents

1 RASP **3**
1.1 The RASP Programming Language 3
1.2 The `Tracr` Compiler 6
1.3 Points of Contrast 7
1.3.1 Functional level: Procedural vs. Declarative Perspectives 7
1.3.2 Algorithmic level: Treatments on the Residual Stream 8
1.3.3 Implementational level: Constraining Boolean Operator Compositions 9
1.3.4 Additional Contribution: the Transformer Explorer Visualization Tool 9

2 QKVL File Description **9**

3 Compiling a PSL program into a QKVL instruction file **13**
3.1 Register Abbreviation 13
3.2 Production Block Processing 13
3.3 Repeat Block Processing 14

4 Tensor Product Representations **14**

5 Exploratory training from scratch and testing **16**
5.1 Training curves 17
5.2 Results by model 21

References **25**

1. RASP

In this appendix, we summarize key insights and features of the Restricted Access Sequence Processing (RASP) language and other related work in the same vein as the TPF framework presented here.

First developed by Weiss, Goldberg, and Yahav, RASP was introduced as a computational model of the transformers. Instead of neural network primitives, the RASP language uses sequence operations as primitives that are conceptually aligned with transformer components. Weiss et al. demonstrated that their handed coded RASP programs are able to solve several sequence-manipulation tasks, but were unable to prove their realizability in transformers. To address this limitation, Lindner, Kramár, Farquhar, Rahtz, McGrath, and Mikulik proposed a compiler named `Tracr` that compiles (a subset of) RASP programs into transformer models, thereby proving that for a given program a corresponding model exists which can implement it. These broadly correspond to the PSL language and the realization of its programs as DAT models presented in this work.

Although conceptually related RASP and PSL differ in some important respects, which end up having important downstream consequences. The most important of these lies in the choice of atomic **atomic data structure** that underlies each approach. At a high level this leads to the following differences at multiple interrelated levels of analysis:

- Functional level: RASP takes a *procedural* perspective on representing transformer computation, while PSL takes a *declarative* perspective. This gives rise to computations that incorporate architectural variations beyond the vanilla transformers in PSL, specifically, recurrence and layer looping.
- Algorithmic level: `Tracr` utilizes the residual stream to encode intermediate variables in the computation procedure specified in RASP, while QKVL uses the inherently defined SSS variables as the residual stream, where variables are stored, modified, and passed along the computation dynamics.
- Implementational level: (among abundant differences) `Tracr` has to restrict RASP to avoid arbitrary boolean operation composition on selectors, while DAT does not have such a constraint.

A full overview of the differences is presented in section 1.3. However, for the benefit of the unfamiliar reader we provide a primer on RASP and Tracr in the following subsections.

1.1 The RASP Programming Language

Instead of the TGT considered in the current work, the RASP language is based on the idea of *sequence manipulation*, with the goal of characterizing how a transformer encoder could perform multi-step logical inferences over input expressions. That is, RASP programs capture the transformations on the input sequence as a whole at each step of computation through a transformer encoder. Therefore, RASP intuitively focuses on mapping the primitive transformer components, i.e., attention and feed-forward computations, into primitives in the programming language.

There are two basic types of RASP operations that respectively correspond to the two main components in the transformer architecture: the element-wise operations (corresponding

to the MLP module) and the `select-aggregate` operations (corresponding to the attention module). An input string `abc` is converted into a sequence through the two built-in sequence operators (*s-ops*): `tokens(''abc'') = [''a'', ''b'', ''c'']` and `indices(''abc'') = [0, 1, 2]`. S-ops are functions that map an input string to a sequence of the same length. S-ops can be composed with arithmetic and logical operators (`+, -, %, if, >, <`, etc.): for instance, `(tokens if (indices % 2 == 0) else ''-'')(''hello'') = ''h-l-o''`. Constant values are treated as s-ops as well, with a single value broadcasting across all positions in order to maintain the sequence status: for instance, `length(''abc'') = [3, 3, 3]`. Therefore, the composition of s-ops is conceptualized as element-wise operations in RASP, which matches the intuition behind the MLP layers.

On the other hand, the attention mechanism is conceptualized in RASP as a two step, `select-aggregate` operation. In a standard transformer, attention scores are obtained by the 'QK circuit', which uses the dot product between queries and keys to determine how each position is weighted. The dot product part is captured by the `select` operation, which takes as input a key sequence k, a query sequence q, and a binary predicate p. It outputs a selection matrix named a *selector* that describes whether condition $p(k, q)$ holds for each (k, q) token pair. For instance (reprinted from Weiss et al., 2021):

$$S \equiv \texttt{select}([0,1,2],[1,2,3],<) = \begin{bmatrix} \mathbf{T} & F & F \\ \mathbf{T} & \mathbf{T} & F \\ \mathbf{T} & \mathbf{T} & \mathbf{T} \end{bmatrix}$$

Next, the 'OV circuit' in a transformer produces the output by weighting the symbols in the value vector according to the attention scores. The weighting and production steps are handled by the `aggregate` operation, which takes as input a selector and a sequence; it outputs another sequence that averages for each row of the selector the values of the sequence in its selected columns — the "averaging over the binary values in each selector row" part does the weighting and the "producing the output given the input sequence" part does the value-vector-based production part. For instance, using the selector S above (also from Weiss et al., 2021):

$$\texttt{aggregate}(S, [10, 20, 30]) = [10, 15, 20]$$

Thus, the `select-aggregate` operation composes a key, a query, and a value s-op into an output s-op. Through a combination of element-wise operations and `select-aggregate` operations, a RASP program composes primitive s-ops into a final s-op, which maps the input string into a sequence of the same length. As is mentioned in Section **??**, it is demonstrated that RASP programs could be designed to solve several sequence manipulation tasks such as: reversing, histogram, double-histograms, sorting, ranking by frequency, etc. Here, we walk through the RASP solution to a simple task of reversing (e.g., `reverse(''abcde'')=''edcba''`). The RASP program for the reversing task is presented as follows (reprinted from Figure 4 of Weiss et al., 2021):

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

This program requires a 2-layer transformer with 1 head per layer. The first layer computes line 1 of the program, where the `select-aggregate` mechanism is required to compute the `length` s-op. Specifically, instead of a primitive, `length` is formally defined as `length = 1 / aggregate(select(1,1,==), indicator(indices==0))`. Then, given input string ``abcde'', `indicator(indices==0) = [1, 0, 0, 0, 0]`. It follows that:

$$\texttt{select_all} \equiv \texttt{select}(1, 1, ==) = \begin{bmatrix} \mathbf{T} & \mathbf{T} & \mathbf{T} & \mathbf{T} & \mathbf{T} \\ \mathbf{T} & \mathbf{T} & \mathbf{T} & \mathbf{T} & \mathbf{T} \\ \mathbf{T} & \mathbf{T} & \mathbf{T} & \mathbf{T} & \mathbf{T} \\ \mathbf{T} & \mathbf{T} & \mathbf{T} & \mathbf{T} & \mathbf{T} \\ \mathbf{T} & \mathbf{T} & \mathbf{T} & \mathbf{T} & \mathbf{T} \\ \mathbf{T} & \mathbf{T} & \mathbf{T} & \mathbf{T} & \mathbf{T} \end{bmatrix}$$

The `select-aggregate` attention operation in layer 1 thus returns an s-op where each entry is the reciprocal of the input length:

$$\texttt{aggregate}(\texttt{select_all}, \texttt{indicator(indices==0)}) = [0.2, 0.2, 0.2, 0.2, 0.2]$$

The following two element-wise operations are then applied on the output of `aggregate` to compute the `opp_index`. They are composed into a single MLP layer, and the output s-op is stored in the residual stream for later use.

- Taking the reciprocal on the output of the attention block: `length = 1 / [0.2, 0.2, 0.2, 0.2, 0.2] = [5, 5, 5, 5, 5]`;
- Get a reversed s-op of the `indices`: `opp_index = length - indices - 1 = [5, 5, 5, 5, 5] - [0, 1, 2, 3, 4] = [1, 1, 1, 1, 1] = [4, 3, 2, 1, 0]`;

The second layer computes line 2 and 3 of the program, where line 2 specifies the selector and line 3 specifies the `aggregate` operation. No element-wise operations are needed at this layer. Given the s-op representing the reversed indices, it is natural to define the attention pattern that selects input characters in the reversed order:

$$\texttt{flip} \equiv \texttt{select(indices, opp_index,==)} = \begin{bmatrix} F & F & F & F & \mathbf{T} \\ F & F & F & \mathbf{T} & F \\ F & F & \mathbf{T} & F & F \\ F & \mathbf{T} & F & F & F \\ \mathbf{T} & F & F & F & F \end{bmatrix}$$

Then, the final `aggregate` operation applies the `flip` selector on the input s-op to obtain the final output:

$$\texttt{aggregate}(\texttt{flip}, \texttt{tokens}) = \texttt{[e, d, c, b, a]}$$

Notice that each RASP program decides the number of layers and the number of attention heads per layer needed for executing the program. See App. 1.2 for more details.

The following table presents rough functional correspondences between elements in RASP, the standard transformer architecture, and PSL.

RASP	Function	Transformer	PSL
s-op	string → sequence	hidden states encoded in the residual stream	cell states in `SSS`
selector	(s-op k, s-op q, predicate p) → selector	attention matrices	Production Condition
element-wise operations	s-op → s-op	MLP modules	N.A.
`select-aggregate` operations	(selector S, s-op v) → s-op	attention heads	Production (evaluating Condition and taking Action)

1.2 The `Tracr` Compiler

Given the structure of the RASP programming language, the `Tracr` compiler first compiles RASP into an intermediate representation that operates on subspaces of the residual steam, which conceptually corresponds to the QKVL in TPF, and then compiles the intermediate representations into weights and matrices that realize a decoder-only transformer model, which conceptually corresponds to the DAT in TPF. We summarize the steps of compiling RASP programs into the intermediate level representation below.

1. Given a RASP program, create a computational graph with each node representing a RASP expression (either an s-op or a selector) and each edge representing a RASP operation (either element-wise or `select-aggregate`). This graph encodes the dependencies between how s-ops are composed, the order of which is directly translated into attention and MLP block arrangement.

2. Given a pre-determined vocabulary and a maximal sequence length, for each node in the graph, treat it as a variable and infer the set of all possible values it can take. Allocating a subspace of the residual stream (by assigning a set of basis vectors) to this variable, such that every variable has a subspace orthogonal to the subspaces of other variables. This achieves a disentangled residual stream, so that every operation reads from and writes to a dedicated subspace.

3. Given the inferred values in step 2, translate each individual node into a transformer block.

    - `Tracr` supports two types of representations: for a categorical variable, each sequence token is represented as a one-hot vector; for a numerical representation, each sequence token is represented as a scalar value in a one-dimensional space.
    - The element-wise operations are translated into MLP blocks based on manually engineered heuristics, with categorical variables handled by lookup tables and numerical variables handled by piecewise linear approximations: discretizing the range of possible values into buckets (with the granularity chosen to minimize approximation error).
    - The select-aggregate operations are translated into attention blocks. A selector is represented by a W_{QK} matrix, and an aggregate operation is represented by a W_{OV} matrix, with only hard attention and categorical inputs allowed.

4. Translate the entire computational graph into an arrangement of transformer blocks. With the goal of finding the smallest possible model, first find out the total number of layers needed by computing the longest path from input to a given node, which denotes the number of attention and MLP modules needed to compute the represented steps. Then, arrange the nodes into alternating attention and MLP blocks without violating any dependency in the graph.

5. Embed each s-op (both the input s-ops to an operation and the output s-ops from an operation) into its own orthogonal subspace, and construct the residual stream space as the direct sum of all components' input and output spaces.

1.3 Points of Contrast

Despite various similarities between {RASP, `Tracr`} and {PSL, QKVL, DAT}, here we focus on several fundamental points of contrast between the two approaches. The choice of different atomic data structures between RASP and PSL leads to drastic difference in conceptualizing the Transformer computations each language is designed to represent, across multiple levels of analysis. In RASP, the atomic representation unit is a sequence operator (*s-op*), a sequence of symbols with no internal structure within each symbol. The other basic representation in RASP, selectors, are binary matrices built on a pair of *s-op*s together with a predicate. In contrast, as is defined in (40b), each computation step of PSL is represented by a layer / sequence of cell states, where each cell state encodes a discrete set of variables such as `region` (r), `field` (f), `index` (d), in addition to `position` (p) and `symbol` (s). That is, PSL carries a richer data structure compared to RASP, as each cell in a PSL state contains multiple registers in addition to the symbol information, while each unit in the s-op sequence carries only the symbol information. The richness of information in PSL, represented by the state-structure space (SSS), induces the following distinctions compared to RASP.

1.3.1 Functional level: Procedural vs. Declarative Perspectives

In RASP, the two types of basic operations, element-wise operations and select-aggregate operations, are defined over *s-op*s that represent sequence-wise computation: transforming an *s-op* (and a selector) into another *s-op*. This sequence manipulation nature of RASP programs displays a procedural perspective: instead of targeting specific positions or symbols in a sequence, RASP specifies computations / transformations on entire sequences, as there is no inherently built-in operations on a more fine-grained level. In compilation, `Tracr` also follows this perspective by tracing a RASP program into a computational graph, where nodes are intermediate *s-op* representations and edges are sequence-transforming operations. The nature of variables associating to specific nodes in the whole computational graph reflects its procedural character.

On the other hand, a PSL program consists of a sequence of productions, where each production is specified by a *Condition* and an *Action*. Although the sequence of productions still specifies the procedure of how PARSE and GEN should take place at the higher level, the variables encoded in each cell state at every step of computation are associated only to individual symbol positions to support for production condition checking and action application, instead of being sensitive to the computation steps in the whole program.

That is, those variables are static properties of the symbols, independent from the step of production the machine is conducting. This reflects the declarative perspective PSL takes on how atomic information is stored and represented.

Moreover, PSL allows for additional computations that incorporate architectural variations beyond the vanilla Transformer, which RASP cannot achieve by design. PSL allows for features other than *symbol* and *position* to persist across timesteps and positions i.e, the layer-1 state of cell $P+1$ can be set to be the layer-L state of cell P. This ability of representing recurrence enables PSL to model computations representable by architectures like the feedback transformer (Fan, Lavril, Grave, Joulin, & Sukhbaatar, 2020) and the staircase transformer (Ju, Roller, Sukhbaatar, & Weston, 2022). Another built-in property of PSL is that it allows for layer looping: repeated application of particular layers. Thus, PSL can also model computations representable by universal transformers (Dehghani, Gouws, Vinyals, Uszkoreit, & Kaiser, 2018) and MOE-Universal transformers (Csordás, Irie, Schmidhuber, Potts, & Manning, 2024). RASP, on the other hand, cannot represent such computations.

1.3.2 Algorithmic level: Treatments on the Residual Stream

After tracing a RASP program into a computational graph, `Tracr` allocates subspaces in the residual stream for storing all nodes in the graph, i.e., all the intermediate *s-op*s needed for later steps of computation. This treatment of the residual stream again reflects the procedural perspective RASP and `Tracr` take: the residual stream is used as the storage space for reading and writing intermediate variables at the representation unit of sequences, and the what has been written into the subspaces reflect the computational traces or activation states the program goes through in executing the program.

Similar to RASP, the residual stream in PSL is also partitioned into subspaces that encode individual variables (or registers, features, attributes) at each position. However, unlike the sequence-wise nature the RASP framework takes, each production in PSL can simultaneously retrieve and check multiple variables at various positions and modify the values of variables targeting specific positions. Accessing and modifying a single variable at any step of the computation is independent from which step the program is current executing, which again reflects the declarative perspective that PSL takes on encoding individual variable information.

RASP could only read and write the content of full subspaces that encode the entire *s-op*s. Thus, each selector, i.e., attention pattern, could only represent the pairwise comparison of two attributes represented by the two *s-op*s it takes. However, the cell state structures enables the flexibility for PSL to read and write variables at the level of unit sequence positions, allowing for simultaneous comparison of multiple sets of attributes when constructing a single attention pattern. At the algorithmic level, QKVL further refines the complexity of representation unit by specifying each cell state with five cell attributes: `query, key, value, input`, and `output`, where each attribute itself is a state structure in SSS that encodes the lower level variables, as is specified in (68). This nested structure at each symbol position in QKVM further exhibits the data-container, declarative character that TPF takes. Again, representing information as symbol sequences versus as data containers give rise to the disparity in the fine-grainedness of information access.

1.3.3 Implementational level: Constraining Boolean Operator Compositions

A further distinction comes at the implementational level. Both RASP and PSL allow for compositions of 'selectors' (in RASP) or conditions (in PSL), allowing layers to harness multiple features at once and thereby efficiently represent programs of increasing complexity. However, in terms of realising this functionality, i.e., compiling such programs into a corresponding transformer, the two frameworks differ. Tracr (the compiler for RASP) cannot compile programs containing operator composition. Though Lindner et al. provide some discussion of how this could be achieved in Appendix B of their paper, their compiler does not support such, more complex, operations. On the other hand, PSL programs involving complex compositions of conjunctions and negations of conditions are readily compilable into DATs. This makes representing sophisticated programs far simpler, which is further complemented by the DAT explorer discussed in the next section.

1.3.4 Additional Contribution: the Transformer Explorer Visualization Tool

Besides the conceptual differences discussed above, a further concrete contribution is the visualization tool, Transformer Explorer, that accompanies PSL (Sec.7.2.3 in the main body). Transformer Explorer offers an interactive GUI that enables visualizing the state of the residual stream at each step of computation, given a prompt that represents a TGT input sequence. Specifically, the GUI displays, via symbolic interpretation of activation vectors, each cell state for all positions across all layers, including the changes in values of each register in each cell state. This enables researchers to keep track of feature activations and modifications at each stage of computation, which improves interpretability and is crucial for debugging and understanding more complex programs. To our current knowledge, visualization tools are not available for prior works.

In sum, TPF provides both a theoretical framework and the necessary existence proof of its realizability through compilable models, together with tooling that makes the process easy and interpretable for researchers.

2. QKVL File Description

The QKVL file is in JSON format. At the top level, it consists of a single dictionary containing system maps and an entry for all the productions from the PSL program:

Dictionary Key	Dictionary Value
register_map	A dictionary of register names and their associated short names
constants_map	A dictionary of constant names and their optional associated strings
system_map	A dictionary of system reserved registers and their associated register names
watch_list	A list of registers to be watched (for `dat_explorer`)
weights	A list of blocks (each either a production dictionary or a repeat-dictionary)

See Sec. 6.1.1 for explanation of the term "weights" here: these are symbolic instructions, not numerical weights, but they will later be compiled into numerical weight matrices for implementation in the DAT.

A production-dictionary represents a PSL production. It contains the following dictionary keys and values:

Dictionary Key	Dictionary Value
layer_comment	the comment associated with this block
causal_attn	specifies if causal attention in effect for block
right_match	specifies if right match attention in effect for block
weights	the weights dictionary for this block

A weights-dictionary represents a production's conditions (in the 'q' and 'k' dictionary keys) and its action (in the 'v' dictionary key):

Dictionary Key	Dictionary Value
q	a registers dictionary for the query weights
k	a registers dictionary for the key weights
v	a registers dictionary for the value weights

A registers-dictionary represents destination/source pairs. The dictionary keys are the destination registers and the associated dictionary values are the source registers or constants. Short names (usually the first letter of the register name) are used for both destination and source registers (except "index" and "parse" are respectively abbreviated "d" and "a"). Each register (e.g., "p") can be optionally followed by 1 of 3 register modifiers:

```
- p            (the value of p in the current column "N")
- p*           (the value of p in the previous column "N - 1")
- p‘           (the value of p in the column "n")
- p*‘          (the value of p in the column "n - 1")
```

If the source value is a register, it can be followed by an optional function, for example:

```
- p@pos_increment
```

The currently supported function names are: `@pos_increment`, `@pos_decrement`.

By default, source values in the k- and v-registers dictionaries represent the "==" operator. Source values for the "!=" operator take the form:

```
- ["!=", <register or constant>]
```

Source values for the "in" operator take the form:

```
- ["in", <comma separated list of constants>]
```

Here is a sample of a production dictionary:

```
{
    "layer_comment": "// parse  step 1b. start Cue ",
    "causal_attn": false,
    "right_match": false,
```

```
        "weights": {
            "q": {
                "s‘": "s",
                "p‘": "1",
                "p": "p",
                "a": "a"
            },
            "k": {
                "s": [
                        "in",
                        "-",
                        ".",
                        "A"
                    ],
                "p‘": "p",
                "p": [
                    "!=",
                    "1"
                ],
                "a": "1"
            },
            "v": {
                "r": "CQ",
                "t": "D",
                "f": "FQ"
            }
        }
    }
```

A repeat dictionary represents a the use of the “repeat” keyword in PSL to repeat a group of blocks until the specified value changes. It consists of 3 key/value pairs:

Key	Value
layer_comment	the comment closest to the start of the repeat block
until	the stopping condition for the repeat processing
weights	the list of weight dictionaries for the inner blocks

Currently, the only condition supported for “until” is “NO_CHANGE”, meaning the blocks are repeated until the value of all registers in all columns after processing the last inner block matches their values before executing the first inner block.

Here is a sample of a repeat dictionary:

```
    {
        "layer_comment": "// repeat pre_2a, 2a. propagate XQ rightward",
        "until": {},
```

```
    "weights": [
        {
            "layer_comment": "// parse  step pre_2a. set prev_region",
            "weights": {
                "q": {
                    "p‘": "p@pos_decrement",
                    "a": "a"
                },
                "k": {
                    "p‘": "p",
                    "a": "1"
                },
                "v": {
                    "r*": "r"
                }
            }
        },
        {
            "layer_comment": "// parse  step 2a. propagate XQ",
            "weights": {
                "q": {
                    "r*": "r*",
                    "r": "r",
                    "a": "a",
                    "p": "p"
                },
                "k": {
                    "r*": "XQ",
                    "r": "R",
                    "a": "1",
                    "p": "p"
                },
                "v": {
                    "r": "XQ"
                }
            }
        }
    ]
}
```

3. Compiling a PSL program into a QKVL instruction file

3.1 Register Abbreviation

Register names are translated to their abbreviation using the "registers" map specified in the PSL program. In addition, if the position-index of the register is "`n`", a backquote ("`") is appended to the abbreviation.[1] Also, if "`@function`" is specified after the register index, then "`@function`" is appended to the abbreviation. The constants for initial values, "`R_INIT`" and "`T_INIT`", are abbreviated to "`R`" and "`T`", and the variables named `w_temp` are abbreviated to "`w`" (for $w = x, y, z$).

3.2 Production Block Processing

Each `<production block>` is processed as follows:

1. The comment closest to the beginning of the production block in the PSL is captured.
2. Empty q, k, and v register dictionaries are created.
3. Each condition in the condition list is added to the q and k register dictionaries as follows:
   - The left-hand — target — register name is translated to its abbreviation.
   - The right-hand — source — register name or constant is translated to its abbreviation. If the comparison operator of the condition is "!=", the right-hand abbreviation is translated to the following list of strings: ["!=", `<right abbreviation>`].
   - If the index of the left-hand register is "n" or "*:n", then:
     - The left/target abbreviation becomes the dictionary key, and the right/source abbreviation becomes the dictionary value; they are added to the k-register dictionary. Then, to the q-register dictionary, the left/target abbreviation is added as both the dictionary key and value, stripping off the backquote of the value, if any.
   - Otherwise:
     - The left/target abbreviation becomes the dictionary key, and the right/source abbreviation becomes the dictionary value; they are added to the q-register dictionary. Then, to the k-register dictionary, the left/target abbreviation is added as both the dictionary key and dictionary value, stripping off the backquote of the value, if any.
4. Each assignment in the assignment list is added as a dictionary key/value pair to the v-register dictionary.
   - The left-hand/target register name is translated to its abbreviation.

1. In Production P5b of (99), the PSL Condition "region[n] == XQ, region[N] == CQ" becomes in QKVL query: $\{r : r, r` : \text{XQ}\}$, key: $\{r : \text{CQ}, r` : r\}$. These match when query[N] == key[n], i.e., if and only if $r[N]$ == CQ and $r[n]$ == XQ.

- The right-hand/source register name or constant is translated to its abbreviation.
- Using the left-hand/target abbreviation as the dictionary key and the right-hand/source constant or abbreviation as the dictionary value, the pair is added to the v-dictionary.

5. A weights-dictionary is created to hold the q-, k-, and v-register dictionaries (dictionary keys are "q", "k", and "v", and the dictionary values are the associated register dictionaries).

6. A production-dictionary is created consisting of a "layer_comment" dictionary key/value and a "weights" dictionary key/value (to hold the weights-dictionary).

3.3 Repeat Block Processing

Each repeat block is processed as follows:

1. The comment closest to the start of the repeat block is captured.

2. The "until" condition is captured.

3. Each production block within the repeat scope is processed as described in Sec. 3.2, resulting in a list of production dictionaries.

4. A new repeat-dictionary is created with the following dictionary key/values:

Key	Value
layer_comment	the comment for the repeat block
until	the condition for terminating the repeat process
weights	the list of production-dictionaries

The final result is a list of production and repeat dictionaries that are converted to JSON format and output to a file.

4. Tensor Product Representations

A general method for encoding symbol structures as neural activity vectors is *Tensor Product Representations* (TPRs) (Smolensky, 1987, 1990).[2] Any symbol structure type can be decomposed as a set of *structural roles*, and a particular token of that type is defined by assigning *fillers* to these roles. In the special case relevant to TPF, the state-variable structure (37) is defined by roles which are the state variables, and fillers that are the values that these variables can be assigned. Retaining the notation of the main text of this paper, we can write $\mathsf{r{:}f}$ for the *binding* of the role r to the filler f. The particular state-variable structure S shown in (37) is defined by the bindings $\beta(\mathsf{S}) = \{r : \mathrm{XQ},\ f : \mathrm{FQ},\ s : \mathrm{Q}, ...\}$

Let us group roles/variables by their set of possible fillers/values. The group $\gamma \in 1, \ldots, \Gamma$ contains the n_γ roles $\mathsf{R}^\gamma = \{\mathsf{r}^\gamma_1, \ldots, \mathsf{r}^\gamma_{n_\gamma}\}$, all of which take fillers in the same set, $\mathsf{F}^\gamma =$

2. Related methods, many under the rubric of *Vector Symbolic Architectures* (Kleyko et al., 2022; Schlegel et al., 2022), might well work as well as TPRs here, provided sufficiently accurate means are available for detecting perfect matching for Conditions and perfect value-changing for Actions.

$\{\mathsf{f}_1^\gamma, \ldots, \mathsf{f}_{m_\gamma}^\gamma\}$. (For TGT, one group would be the set of region-valued variables $\{r, r^*, r^\backprime, r^{*\backprime}\}$.) To create a TPR, we adopt row-vector embeddings of roles $\mathbb{E}_\mathrm{R} : \mathsf{R}^\gamma \to \overrightarrow{\mathsf{R}^\gamma}; \mathsf{r}_i^\gamma \mapsto \overrightarrow{\mathsf{r}_i^\gamma}$ and fillers $\mathbb{E}_\mathrm{F} : \mathsf{F}^\gamma \to \overrightarrow{\mathsf{F}^\gamma}; \mathsf{f}_j^\gamma \mapsto \overrightarrow{\mathsf{f}_j^\gamma}$. Then the TPR for a particular symbol structure S defined by the bindings $\beta(\mathsf{S}) = \{\mathsf{r}_i^\gamma : \mathsf{f}_{j(i)}^\gamma \mid \gamma = 1, \ldots, \Gamma;\ i = 1, \ldots, n_\gamma;\ j(i) \in 1, \ldots, m_\gamma\}$ is the tensor $\mathbf{T}_\mathsf{S} \equiv \bigoplus_{\gamma=1}^{\Gamma} \sum_{i=1}^{n_\gamma} \overrightarrow{\mathsf{r}_i^\gamma} \otimes \overrightarrow{\mathsf{f}_{j(i)}^\gamma} \in \bigoplus_{\gamma=1}^{\Gamma} \overrightarrow{\mathsf{R}^\gamma} \otimes \overrightarrow{\mathsf{F}^\gamma} \equiv \mathcal{V}$ or equivalently the matrix $\mathbf{T}_\mathsf{S} \equiv \bigoplus_{\gamma=1}^{\Gamma} \sum_{i=1}^{n_\gamma} \overrightarrow{\mathsf{r}_i^\gamma}^\top \overrightarrow{\mathsf{f}_{j(i)}^\gamma}$. (This 'matrix' is essentially a sequence of Γ matrices, each operating on a different subspace $\overrightarrow{\mathsf{R}^\gamma} \otimes \overrightarrow{\mathsf{F}^\gamma}$ of the overall direct-sum vector space $\mathcal{V}$.)

We now focus on a single subspace defined by a value of $\gamma \in 1, \ldots, \Gamma$, and leave the superscript γ implicit, e.g., writing n for n^γ.

TPRs are designed so that compositional operations on entire structures can be performed in parallel by operating on them with linear transformations (Smolensky, 2012) — but also so that it is possible to accurately extract fillers of individual roles when needed. In the standard case — lossless TPR encoding — the role embeddings are chosen to be linearly independent, which guarantees the existence of a set of *dual vectors* $\{\overrightarrow{\mathsf{r}_i}^+\}$ defined so that $\overrightarrow{\mathsf{r}_i}^+ \cdot \overrightarrow{\mathsf{r}_j} = \delta_{ij}$. Then to unbind role $\overrightarrow{\mathsf{r}_i}$ it suffices to compute $\overrightarrow{\mathsf{r}_i}^+\, \mathbb{T}_\mathsf{S} \equiv \widehat{\overrightarrow{\mathsf{f}_i}}$. It is straightforward to verify that this is an accurate extraction, that $\widehat{\overrightarrow{\mathsf{f}_i}} = \overrightarrow{\mathsf{f}_i}$:

$$\widehat{\overrightarrow{\mathsf{f}_i}} = \overrightarrow{\mathsf{r}_i}^+\, \mathbb{T}_\mathsf{S} = \overrightarrow{\mathsf{r}_i}^+ \left(\sum_{j=1}^n \overrightarrow{\mathsf{r}_j}^\top \overrightarrow{\mathsf{f}_j} \right) = \sum_{j=1}^n \left(\overrightarrow{\mathsf{r}_i}^+ \cdot \overrightarrow{\mathsf{r}_j} \right) \overrightarrow{\mathsf{f}_j} = \sum_{j=1}^n \delta_{ij} \overrightarrow{\mathsf{f}_j} = \overrightarrow{\mathsf{f}_i}$$

This error-free extraction occurs whenever the role embeddings are linearly independent; in the general case, these are dense vectors and the TPR is fully distributed — every element of the tensor contributes to encoding the filler of every role; there are no separable 'registers' for the roles.

Note that in the special case that the linearly independent role embeddings are orthonormal, then $\overrightarrow{\mathsf{r}_i}^+ = \overrightarrow{\mathsf{r}_i}$. This is the fully distributed orthonormal embedding of (69b).

One-hot role embeddings are yet a further special case of orthonormality (since fully dense vectors can also be orthonormal). In this special case, $\mathbb{T}_\mathsf{S}$ is simply the matrix in which the kth row is $\overrightarrow{\mathsf{f}_k}$. For the state-variable structure of TPF, the kth row of $\mathbb{T}_\mathsf{S}$ is exactly the register for the kth state variable, containing the vector $\overrightarrow{\mathsf{f}_k}$ which is the embedding of the value of that variable; this is the semi-local case of (69a). In the yet further special case when the filler embeddings are also 1-hot, this reduces to exactly the fully local embedding of (68), visualized in (67).

We now show that the DAT can implement a PSL program correctly without using the 1-hot, fully local embedding of state-variable structures deployed in the main text (and the current software). We will assume a TPR embedding of state structures, and require only that the role embeddings are orthonormal, and that the filler embeddings are normalized.

To work, what DAT requires of its state-variable-structure representation is only that: for Conditions, we can accurately identify perfectly matching query and key vectors; and for Actions, that we can accurately adjust state variables according to the demands encoded in the value vectors.

As for the Condition requirement, the dot product of the TPRs for two SSS structures S and S' (a query and a key, in the DAT case), when we adopt orthonormal (not necessarily 1-hot) role embeddings, is[3]

$$\mathbf{T}_{\mathsf{S}} \cdot \mathbf{T}_{\mathsf{S}'} = \left(\sum_{i=1}^{M} \overrightarrow{\mathsf{r}_i} \otimes \overrightarrow{\mathsf{f}_i}\right) \cdot \left(\sum_{j=1}^{M} \overrightarrow{\mathsf{r}_j} \otimes \overrightarrow{\mathsf{f}'_j}\right) = \sum_{i=1}^{M} \overrightarrow{\mathsf{f}_i} \cdot \overrightarrow{\mathsf{f}'_i} = \sum_{i=1}^{M} \cos(\overrightarrow{\mathsf{f}_i}, \overrightarrow{\mathsf{f}'_i}) \, \| \overrightarrow{\mathsf{f}_i} \| \, \| \overrightarrow{\mathsf{f}'_i} \|$$

If all filler vectors are normalized, $\| \overrightarrow{\mathsf{f}_i} \| = 1$, then each term in this dot product reduces to $\cos(\overrightarrow{\mathsf{f}_i}, \overrightarrow{\mathsf{f}'_i})$, which is ≤ 1, with equality holding only when $\overrightarrow{\mathsf{f}_i} = \overrightarrow{\mathsf{f}'_i}$. Let the total number of non-null values (non-zero filler vectors) in the query be m.[4] Then dividing the dot product by m gives a value less than 1 unless $\overrightarrow{\mathsf{f}_i} = \overrightarrow{\mathsf{f}'_i}$ for all roles i with non-zero fillers. So let us define the query and key vectors $\mathbf{q}$ and $\mathbf{k}$ to be the TPRs for their respective defining state-variable structures S and S' in SSS, divided by $\sqrt{m}$. Then their dot product is

$$\mathbf{q} \cdot \mathbf{k} = (\mathbf{T}_{\mathsf{S}}/\sqrt{m}) \cdot (\mathbf{T}_{\mathsf{S}'}/\sqrt{m}) = (\mathbf{T}_{\mathsf{S}} \cdot \mathbf{T}_{\mathsf{S}'})\,/m$$

So just as for the 1-hot encoding in the body of the paper (70c-i), requiring this dot product to be 1 enforces perfect matching of all non-null variable values specified in the query.

For implementing a production's Action, to set the value of variable r in the TPR residual stream $\mathbb{O}$ to f, we use

$$\mathbb{O} \mapsto \mathbb{O} + \overrightarrow{\mathsf{r}}^{\top} \left[\overrightarrow{\mathsf{f}} - \overrightarrow{\mathsf{r}}^{+} \mathbb{O} \right]$$

In this adjustment, the first term inserts the value f into r while the second removes the existing value of r (if any).[5]

Since the DAT could function perfectly well with non-1-hot encoding of state variables, given a trained transformer capable of performing ICL, it is entirely possible that it is performing computation quite similar to our DAT, using distributed (dense) encodings. Testing this hypothesis can be pursued through the steps detailed in Sec. 9.2.

5. Exploratory training from scratch and testing

This appendix includes the results for models trained from scratch on the TGT dataset and then tested on various in and out of distribution splits.

The following sequence to sequence models were tested: vanilla transformer (encoder/decoder), nano_gpt (decoder only), nano_gpt_attn_only (no MLP layers), cnn, lstm_attn (LSTM with attention), and mamba.

Here are the architecture hyperparameters:

3. Here we have used the identity $(\overrightarrow{u} \otimes \overrightarrow{v}) \cdot (\overrightarrow{x} \otimes \overrightarrow{y}) = (\overrightarrow{u} \cdot \overrightarrow{x})(\overrightarrow{v} \cdot \overrightarrow{y})$ as well as the orthonormality condition on roles $\overrightarrow{\mathsf{r}_i} \cdot \overrightarrow{\mathsf{r}_j} = \delta_{ij}$
4. As noted in note 12 of the main text, for the DAT (i.e., transformer weights compiled from a PSL program), the set of state-variables assigned non-null values is always the same for query and key, thus avoiding any match issue arising from variables that are null-valued in one but not the other.
5. Instead, it would also be possible to adapt the approach used in the main text (70c-iv), adding to the old value of the variable an encoding of the new value up-weighted by κ, and then applying DATnorm to eliminate all but the most-active value. In the non-1-hot case, DATnorm is defined by extracting the filler vector (value) of each role vector (variable), finding the closest element in the filler-vector dictionary, and setting the variable to that value, as shown above, removing the current value. This version of DATnorm may be useful for other purposes, but for applying the Action of a production, the method proposed in the text of this Appendix is clearly simpler.

Model	hidden	filter	layers	heads	state size	bidir
transformer	512	512	3 + 3	1		
nano_gpt	512	512	6	1		
nano_gpt_attn_only	512	512	6	1		
cnn	512	512	3 + 3			
lstm_attn	512		3 + 3			false
mamba	512	512	18		16	

Table 1: Architecture hyperparameters.

Here are the training hyperparameters:

Model	LR	weight decay	steps	early stop	batch size	dropout
transformer	.0001	0	120,000	false	128	0
nano_gpt	.0001	0	120,000	false	256	0
nano_gpt_attn_only	.0001	0	120,000	false	256	0
cnn	.0001	0	120,000	false	128	0
lstm_attn	.0001	0	120,000	false	128	0
mamba	.0001	0	120,000	false	256	0

Table 2: Training hyperparameters

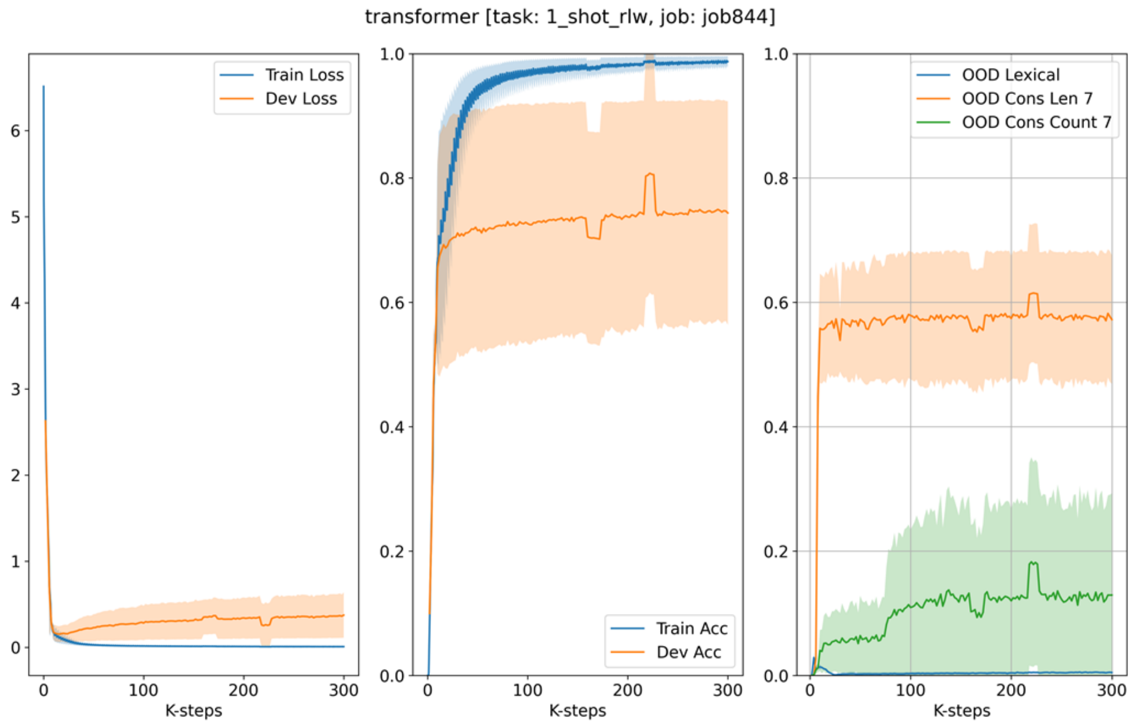
The following splits were tested: train, dev, ood lexical (out of distribution for constituent part vocabulary), ood cons len 7 (constituent lengths of 7), and ood cons count 7 (7 constituents).

All reported metrics were averaged over 3 runs.

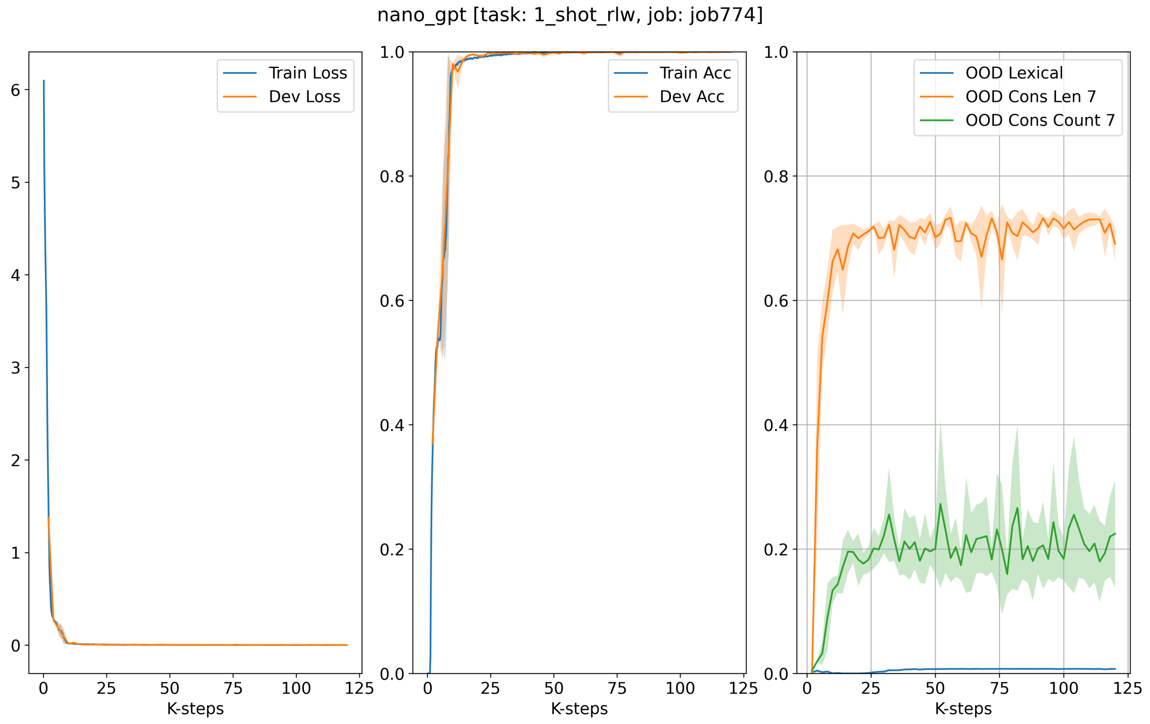
5.1 Training curves

Here are the training curves for each model that we trained.

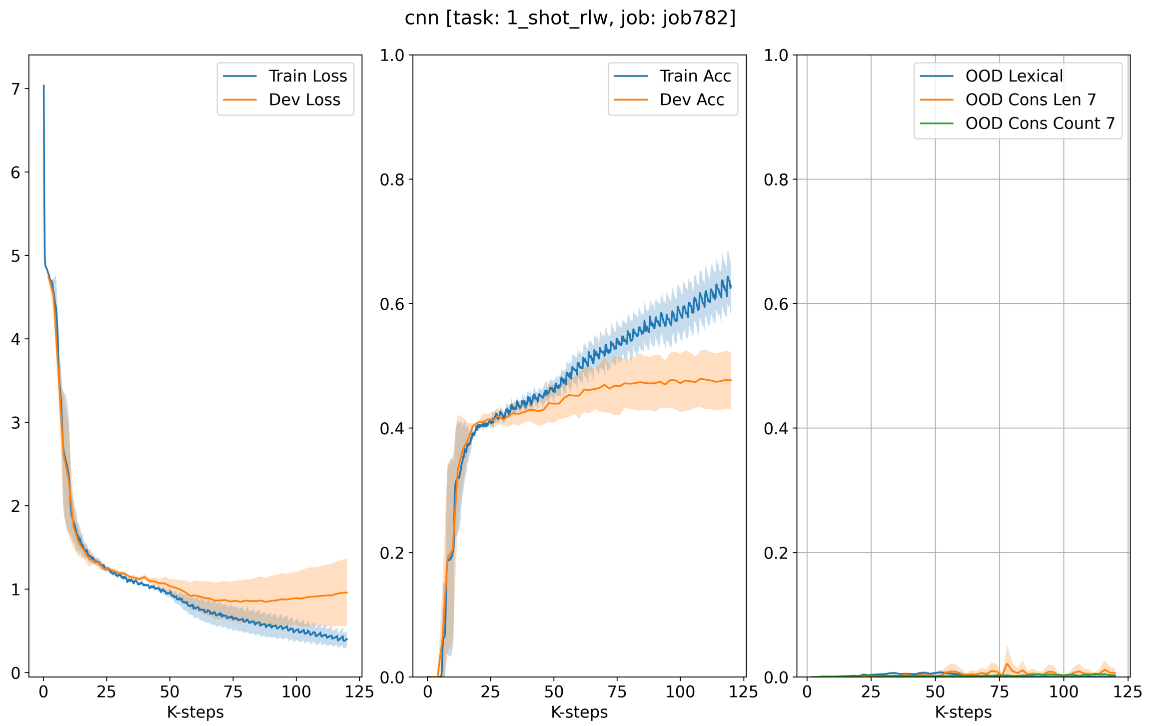
Transformer



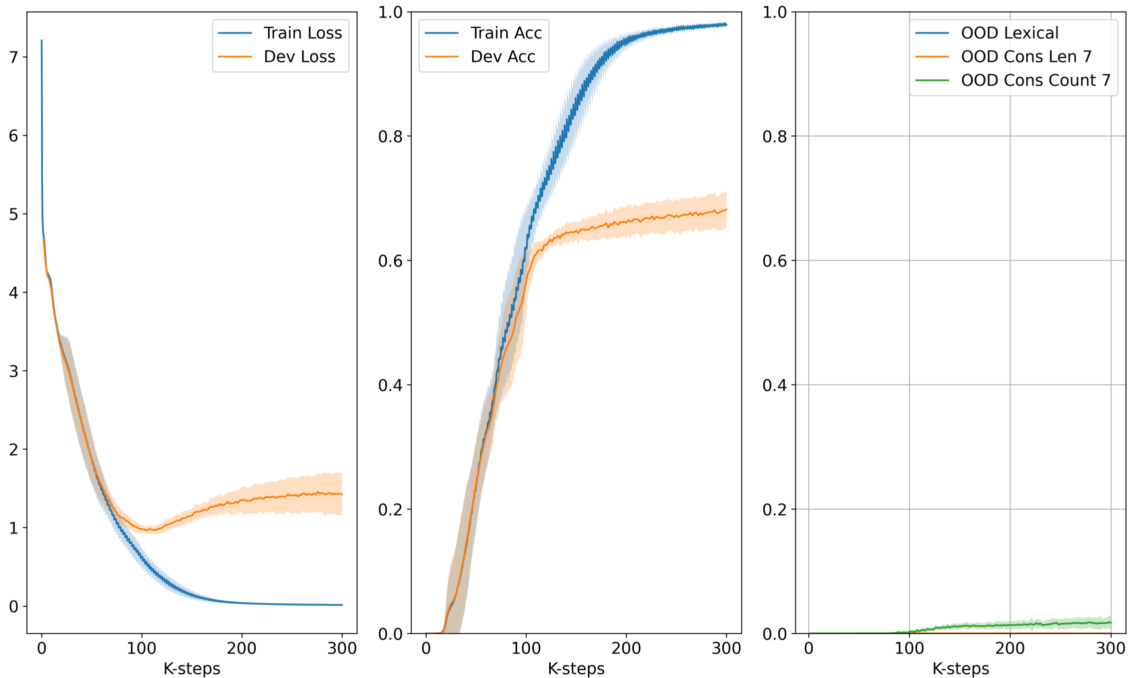
Nano GPT



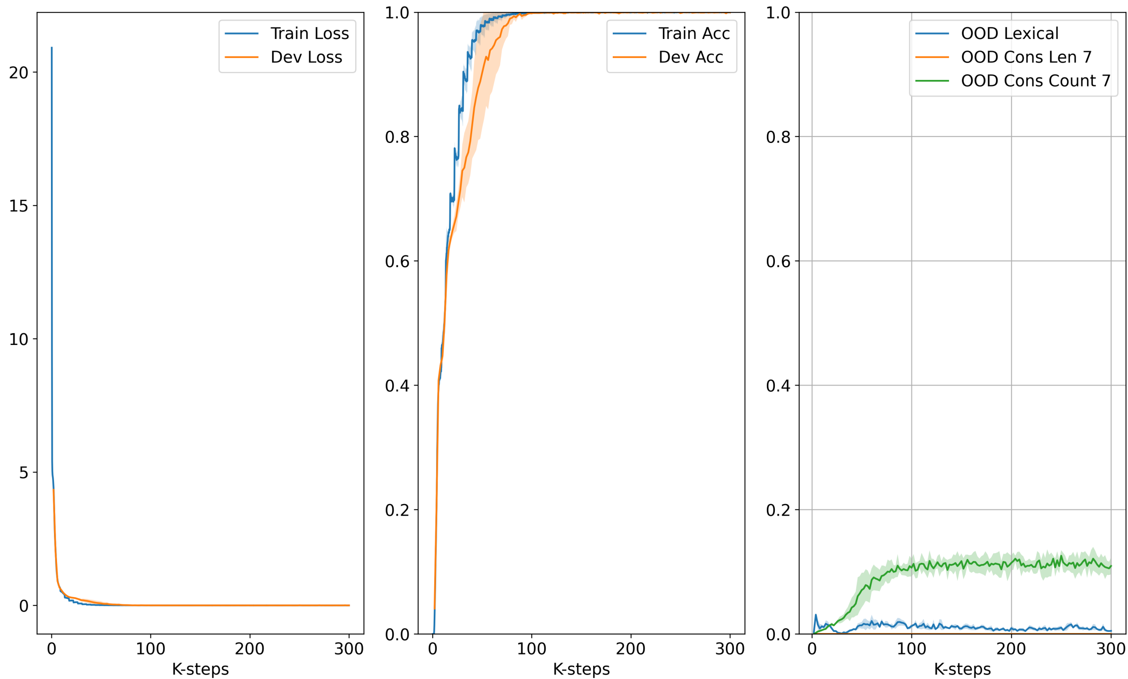
CNN



LSTM with attention



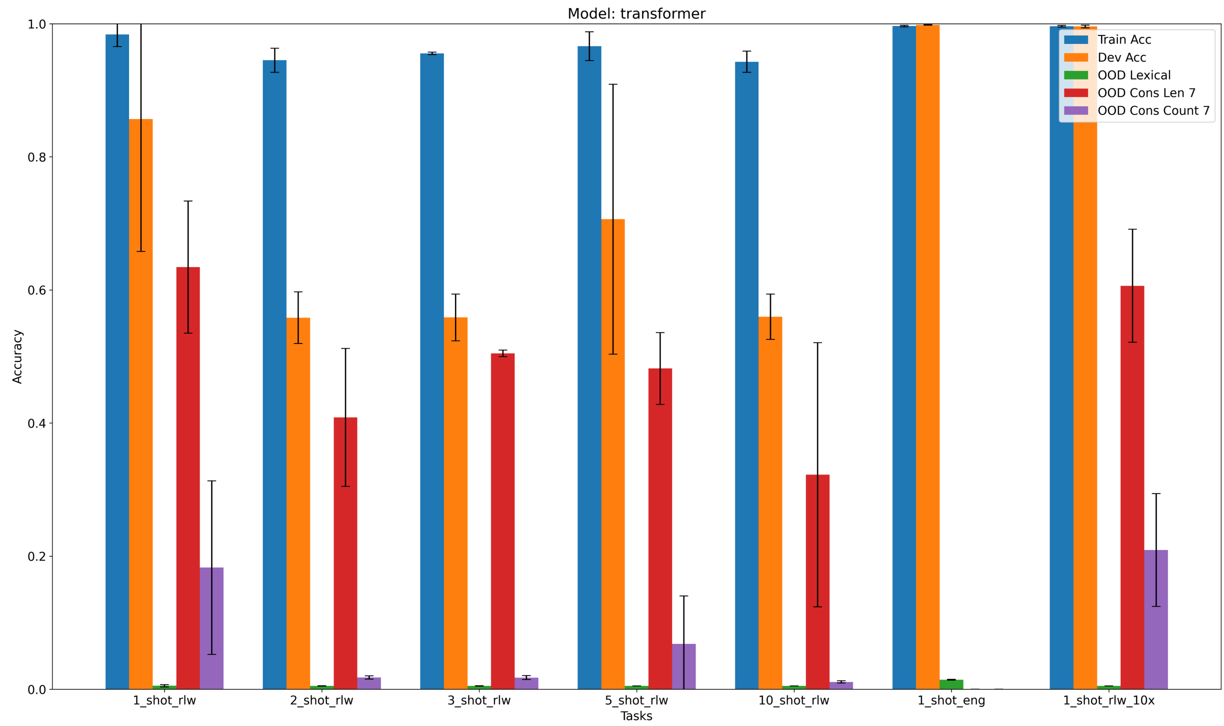
Mamba



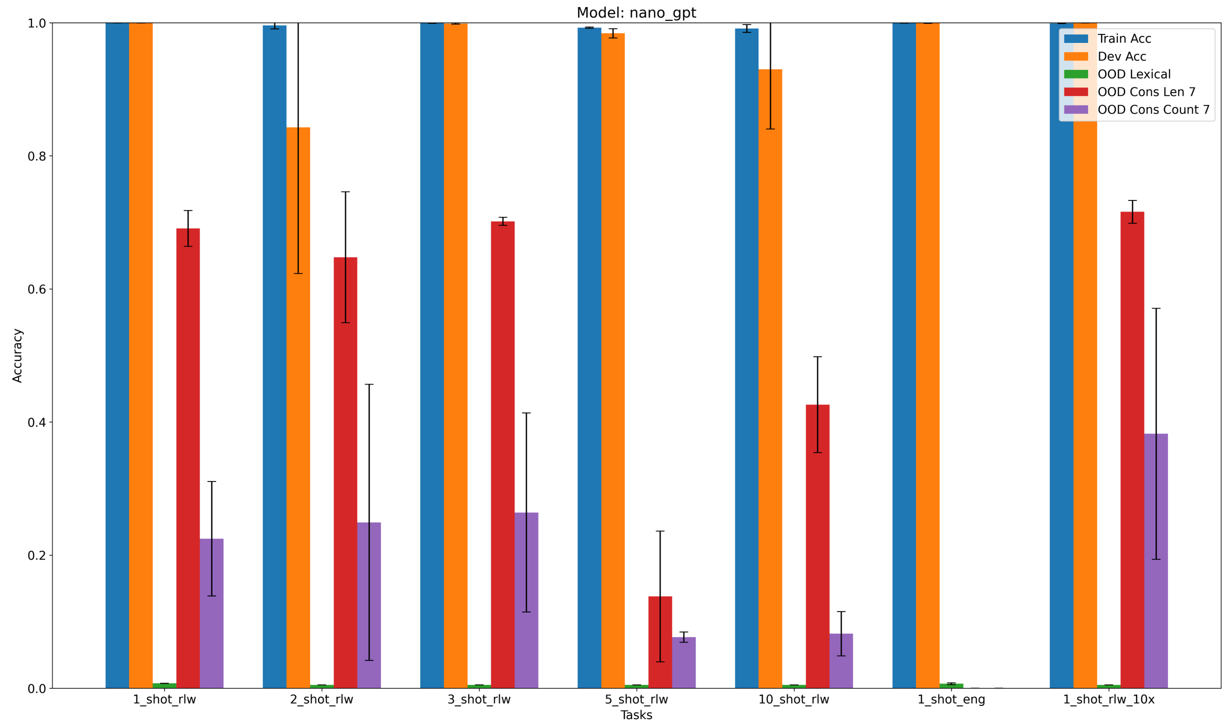
5.2 Results by model

Here are the training results, plotted by model.

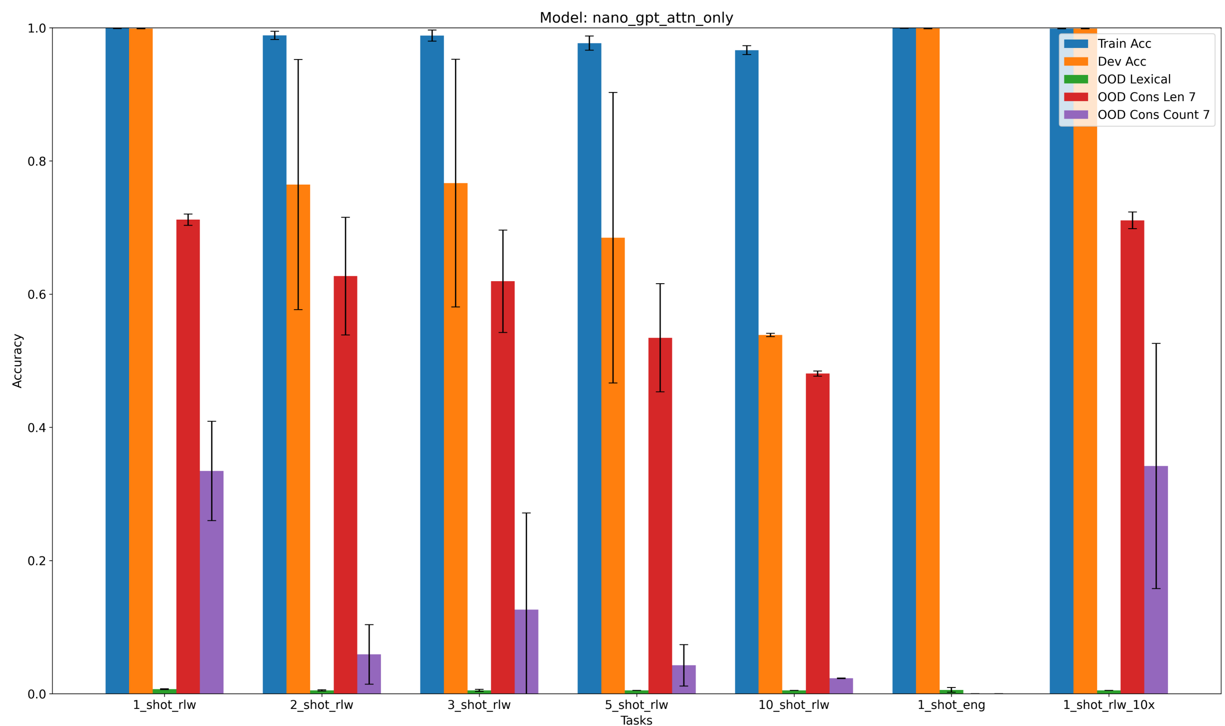
Transformer



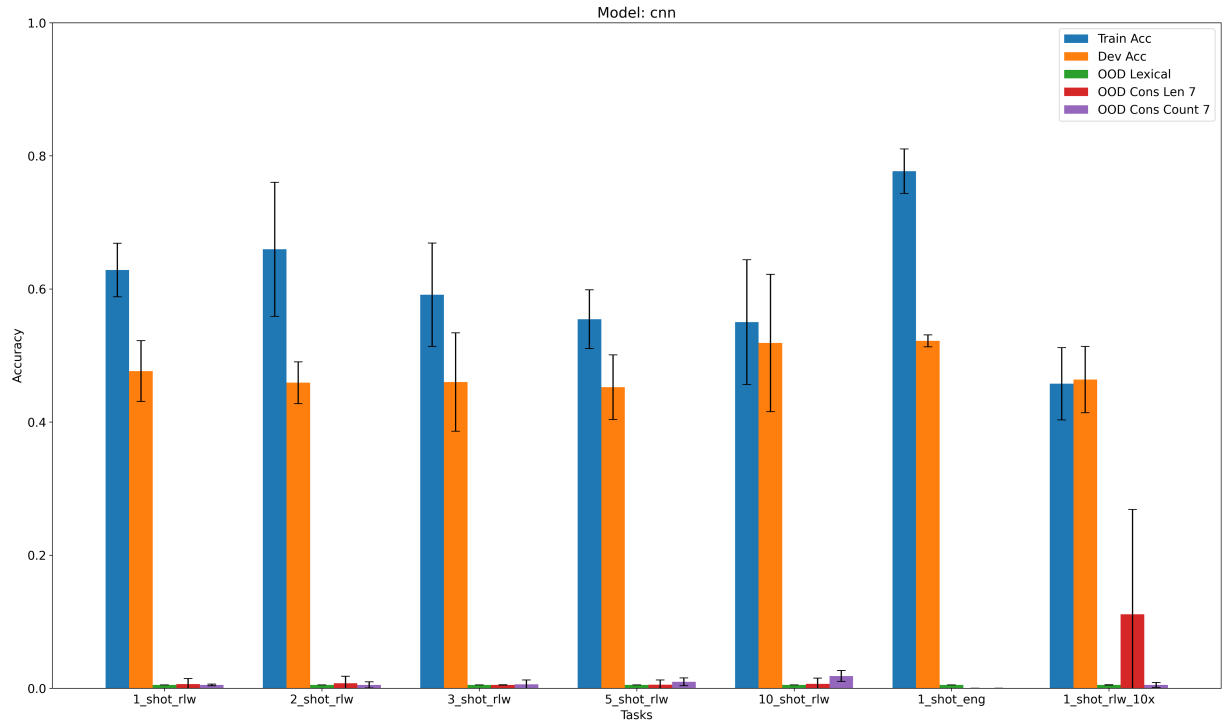
Nano GPT



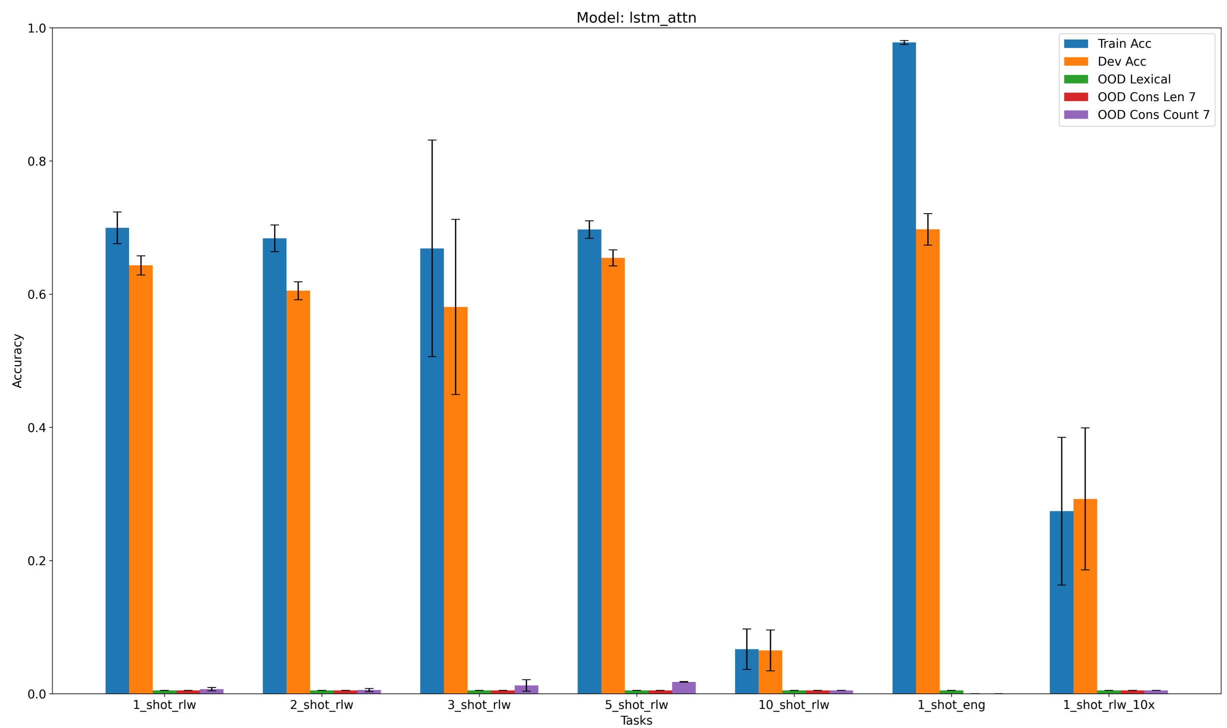
Nano GPT with no MLP layers



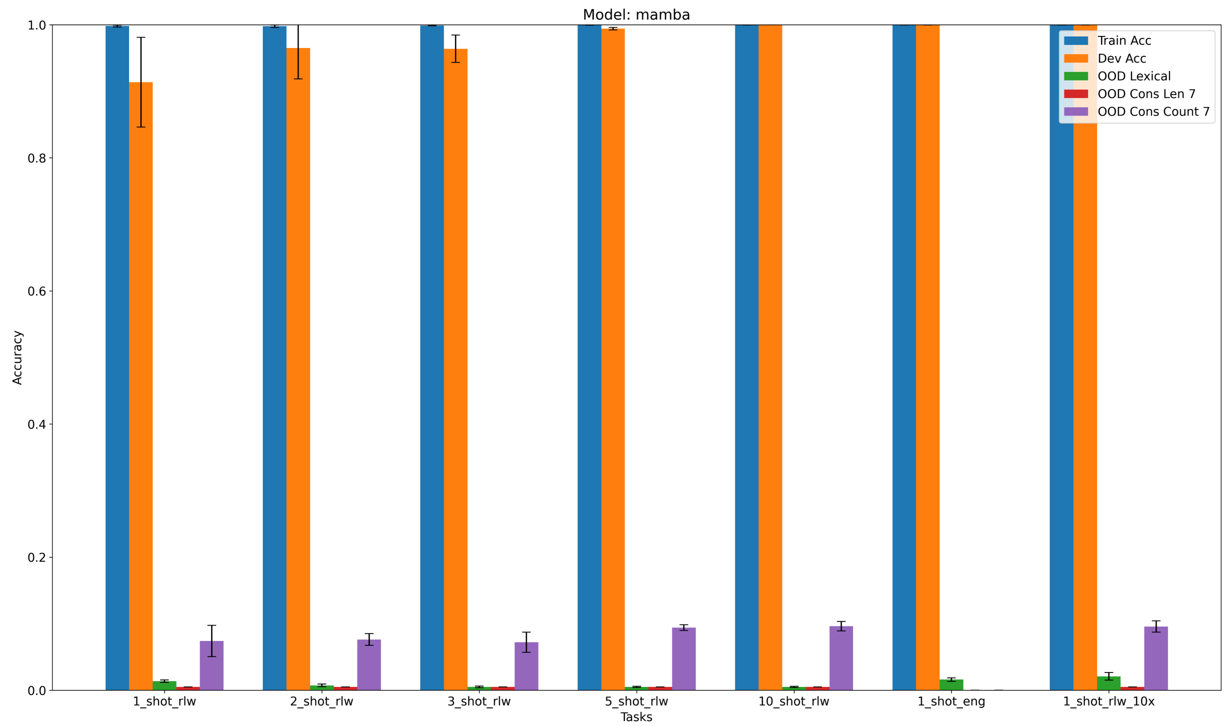
CNN



LSTM with attention



Mamba



References

Csordás, R., Irie, K., Schmidhuber, J., Potts, C., & Manning, C. D. (2024). Moeut: Mixture-of-experts universal transformers. *Advances in Neural Information Processing Systems*, *37*, 28589–28614.

Dehghani, M., Gouws, S., Vinyals, O., Uszkoreit, J., & Kaiser, Ł. (2018). Universal transformers. *arXiv preprint arXiv:1807.03819*.

Fan, A., Lavril, T., Grave, E., Joulin, A., & Sukhbaatar, S. (2020). Addressing some limitations of transformers with feedback memory. *arXiv preprint arXiv:2002.09402*.

Ju, D., Roller, S., Sukhbaatar, S., & Weston, J. E. (2022). Staircase attention for recurrent processing of sequences. *Advances in neural information processing systems*, *35*, 13203–13213.

Kleyko, D., Rachkovskij, D. A., Osipov, E., & Rahimi, A. (2022). A survey on hyperdimensional computing aka vector symbolic architectures, Part I: Models and data transformations. *ACM Computing Surveys*, *55*(6), 1–40.

Lindner, D., Kramár, J., Farquhar, S., Rahtz, M., McGrath, T., & Mikulik, V. (2023). Tracr: Compiled transformers as a laboratory for interpretability. In Oh, A., Naumann, T., Globerson, A., Saenko, K., Hardt, M., & Levine, S. (Eds.), *Advances in Neural Information Processing Systems*, Vol. 36, pp. 37876–37899. Curran Associates, Inc.

Schlegel, K., Neubert, P., & Protzel, P. (2022). A comparison of vector symbolic architectures. *Artificial Intelligence Review*, *55*(6), 4523–4555.

Smolensky, P. (1987). Analysis of distributed representation of constituent structure in connectionist systems. In *Proceedings of the 1987 International Conference on Neural Information Processing Systems*, pp. 730–739.

Smolensky, P. (1990). Tensor product variable binding and the representation of symbolic structures in connectionist systems. *Artificial Intelligence*, *46*(1-2), 159–216.

Smolensky, P. (2012). Symbolic functions from neural computation. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, *370*(1971), 3543–3569.

Weiss, G., Goldberg, Y., & Yahav, E. (2021). Thinking like transformers. *arXiv preprint arXiv:2106.06981*.