Thinking Like Transformers

Gail Weiss [1] **Yoav Goldberg** [2 3] **Eran Yahav** [1]

Abstract

What is the computational model behind a Transformer? Where recurrent neural networks have direct parallels in finite state machines, allowing clear discussion and thought around architecture variants or trained models, Transformers have no such familiar parallel. In this paper we aim to change that, proposing a computational model for the transformer-encoder in the form of a programming language. We map the basic components of a transformer-encoder—attention and feed-forward computation—into simple primitives, around which we form a programming language: the Restricted Access Sequence Processing Language (RASP). We show how RASP can be used to program solutions to tasks that could conceivably be learned by a Transformer, and how a Transformer can be trained to mimic a RASP solution. In particular, we provide RASP programs for histograms, sorting, and Dyck-languages. We further use our model to relate their difficulty in terms of the number of required layers and attention heads: analyzing a RASP program implies a maximum number of heads and layers necessary to encode a task in a transformer. Finally, we see how insights gained from our abstraction might be used to explain phenomena seen in recent works.

1. Introduction

We present a *computational model for the transformer architecture* in the form of a simple language which we dub RASP (*Restricted Access Sequence Processing Language*). Much as the token-by-token processing of RNNs can be conceptualized as finite state automata (Cleeremans et al., 1989), our language captures the unique information-flow constraints under which a transformer operates as it processes input sequences. Our model helps reason about how a transformer operates at a higher-level of abstraction, reasoning in terms of a composition of *sequence operations* rather than neural network primitives.

We are inspired by the use of automata as an abstract computational model for recurrent neural networks (RNNs). Using automata as an abstraction for RNNs has enabled a long line of work, including extraction of automata from RNNs (Omlin & Giles, 1996; Weiss et al., 2018b; Ayache et al., 2018), analysis of RNNs' practical expressive power in terms of automata (Weiss et al., 2018a; Rabusseau et al., 2019; Merrill, 2019; Merrill et al., 2020b), and even augmentations based on automata variants (Joulin & Mikolov, 2015). Previous work on transformers explores their computational power, but does not provide a computational model (Yun et al., 2020; Hahn, 2020; Pérez et al., 2021).

Thinking in terms of the RASP model can help derive computational results. Bhattamishra et al. (2020) and Ebrahimi et al. (2020) explore the ability of transformers to recognize Dyck-k languages, with Bhattamishra et al. providing a construction by which Transformer-encoders can recognize a simplified variant of Dyck-k. Using RASP, we succinctly express the construction of (Bhattamishra et al., 2020) as a short program, and further improve it to show, for the first time, that transformers can fully recognize Dyck-k for all k.

Scaling up the complexity, Clark et al. (2020) showed empirically that transformer networks can learn to perform multi-step logical reasoning over first order logical formulas provided as input, resulting in "soft theorem provers". For this task, the mechanism of the computation remained elusive: how does a transformer perform even non-soft theorem proving? As the famous saying by Richard Feynman goes, "what I cannot create, I do not understand": using RASP, we were able to write a program that performs similar logical inferences over input expressions, and then "compile" it to the transformer hardware, defining a sequence of attention and multi-layer perceptron (MLP) operations.

Considering computation problems and their implementations in RASP allows us to "think like a transformer" while abstracting away the technical details of a neural network in favor of symbolic programs. Recognizing that a task is representable in a transformer is as simple as finding a RASP program for it, and communicating this solution—previously done by presenting a hand-crafted transformer

[1]Technion, Haifa, Israel [2]Bar Ilan University, Ramat Gan, Israel [3]Allen Institute for AI. Correspondence to: Gail Weiss <sgailw@cs.technion.ac.il>.

Proceedings of the 38^{th} International Conference on Machine Learning, PMLR 139, 2021.

```
same_tok = select(tokens,tokens,==);
hist = selector_width(
            same_tok,
            assume_bos = True);

first = not has_prev(tokens);
same_count = select(hist,hist,==);
same_count_reprs = same_count and
    select(first,True,==);

hist2 = selector_width(
            same_count_reprs,
            assume_bos = True);
```
(*a*)

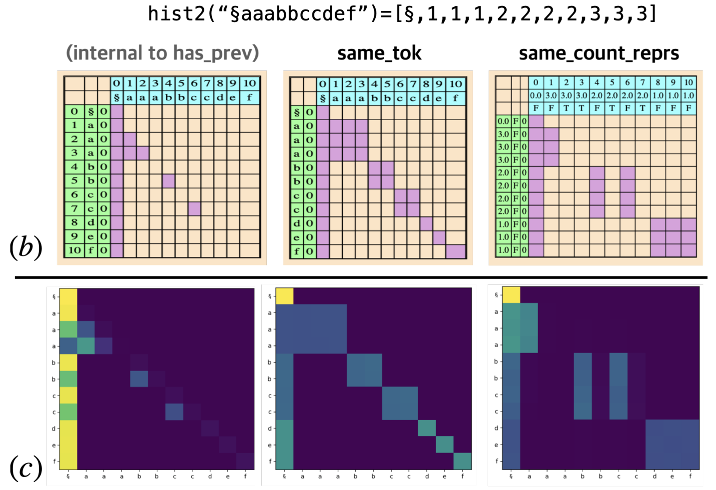

Figure 1: We consider *double-histogram*, the task of counting for each input token how many unique input tokens have the same frequency as itself (e.g.: `hist2("§aaabbccdef")=[§,1,1,1,2,2,2,2,3,3,3]`). (a) shows a RASP program for this task, (b) shows the selection patterns of that same program, compiled to a transformer architecture and applied to the input sequence `§aaabbccdef`, (c) shows the corresponding attention heatmaps, for the same input sequence, in a 2-layer 2-head transformer trained on double-histogram. This particular transformer was trained using both *target* and *attention* supervision, i.e.: in addition to the standard cross entropy loss on the target output, the model was given an MSE-loss on the difference between its attention heatmaps and those expected by the RASP solution. The transformer reached test accuracy of 99.9% on the task, and comparing the selection patterns in (b) with the heatmaps in (c) suggests that it has also successfully learned to replicate the solution described in (a).

for the task—is now possible through a few lines of code. Thinking in terms of RASP also allows us to shed light on a recent empirical observation of transformer variants (Press et al., 2020), and to find concrete limitations of "efficient transformers" with restricted attention (Tay et al., 2020).

In Section 5, we show how a compiled RASP program can indeed be realised in a neural transformer (as in Figure 1), and occasionally is even the solution found by a transformer trained on the task using gradient descent (Figs 5 and 4).

Code We provide a RASP read-evaluate-print-loop (REPL) in `http://github.com/tech-srl/RASP`, along with a RASP cheat sheet and link to replication code for our work.

2. Overview

We begin with an informal overview of RASP, with examples. The formal introduction is given in Section 3.

Intuitively, transformers' computations are applied to their entire input in parallel, using attention to draw on and combine tokens from several positions at a time as they make their calculations (Vaswani et al., 2017; Bahdanau et al., 2015; Luong et al., 2015). The iterative process of a transformer is then not along the length of the input sequence but rather the depth of the computation: the number of layers it applies to its input as it works towards its final result.

The computational model. Conceptually, a RASP computation over length-n input involves manipulation of sequences of length n, and matrices of size $n \times n$. There are no sequences or matrices of different sizes in a RASP computation. The abstract computation model is as follows:

The input of a RASP computation is two sequences, *tokens* and *indices*. The first contains the user-provided input, and the second contains the range $0, 1, ..., n-1$. The output of a RASP computation is a sequence, and the consumer of the output can choose to look only at specific output locations.

Sequences can be transformed into other sequences through element-wise operations. For example, for the sequences $s_1 = [1, 2, 3]$ and $s_2 = [4, 5, 6]$, we can derive $s_1 + s_2 = [5, 7, 9]$, $s_1 + 2 = [3, 4, 5]$, $pow(s_1, 2) = [1, 4, 9]$, $s_1 > 2 = [F, F, T]$, pairwise_mul$(s_1, s_2) = [4, 10, 18]$, and so on.

Sequences can also be transformed using a pair of *select* and *aggregate* operations (Figure 2). Select operations take two sequences k, q and a boolean predicate p over pairs of values, and return a *selection matrix* S such that for every $i, j \in [n]$, $S_{[i][j]} = p(k_{[i]}, q_{[j]})$. Aggregate operations take a matrix S and a numeric sequence v, and return a sequence s in which each position $s_{[i]}$ combines the values in v according to row i in S (see full definition in Section 3).

Aggregate operations (over select matrices) are the only way to combine values from different sequence positions, or to

```
s = select([1,2,2],[0,1,2],==)    res=aggregate(s, [4,6,8])

     1 2 2                  4 6 8
   0 F F F          F F F  4 6 8  => 0
   1 T F F          T F F  4 6 8  => 4  =>  [0,4,7]
   2 F T T          F T T  4 6 8  => 7
```

Figure 2: Visualizing the select and aggregate operations. On the left, a selection matrix **s** is computed by select, which marks for each **query** position all of the **key** positions with matching values according to the given comparison operator ==. On the right, aggregate uses **s** as a filter over its input **values**, averaging only the selected **values** at each position in order to create its output, **res**. Where no **values** have been selected, aggregate substitutes 0 in its output.

move values from one position to another. For example, to perform the python computation: `x = [a[0] for _ in a]`, we must first use $S = \text{select}(indices, 0, =)$ to select the first position, and then $x = \text{aggregate}(S, a)$ to broadcast it across a new sequence of the same length.

RASP programs are lazy functional, and thus operate on functions rather than sequences. That is, instead of a sequence indices$= [0, 1, 2]$, we have a function `indices` that returns $[0, 1, 2]$ on inputs of length 3. Similarly, `s3=s1+s2` is a function, that when applied to an input x will produce the value s3(x), which will be computed as s1(x)+s2(x). We call these functions *s-ops (sequence operators)*. The same is true for the selection matrices, whose functions we refer to as *selectors*, and the RASP language is defined in terms of s-ops and selectors, not sequences and matrices. However, the conceptual model to bear in mind is that of operations over sequences and selection matrices.

Example: Double Histograms The RASP program in Figure 1 solves *double-histogram*, the task of counting for each token how many unique input tokens in the sequence have the same frequency as its own: `hist2("§aabcd")=[§,1,1,3,3,3]`. The program begins by creating the the selector `same_tok`, in which each input position focuses on all other positions containing the same token as its own, and then applies the RASP operation `selector_width` to it in order to obtain the s-op `hist`, which computes the frequency of each token in the input: `hist("hello")=[1,1,2,2,1]`. Next, the program uses the function `has_prev`[1] to create the s-op `first`, which marks the first appearance of each token in a sequence: `first("hello")=[T,T,T,F,T]`. Finally, applying `selector_width` to the selector `same_count_reprs`, which focuses each position on all 'first' tokens with the same frequency as its own, provides `hist2` as desired.

[1]Presented in Figure 12 in Appendix B.

```
1  def frac_prevs(sop,val){
2    prevs = select(indices,indices,<=);
3    return aggregate(prevs,
4                 indicator(sop==val));
5  }
6
7  def pair_balance(open,close) {
8    opens = frac_prevs(tokens,open);
9    closes = frac_prevs(tokens,close);
10   return opens - closes;
11 }
12
13 bal1 = pair_balance("(",")");
14 bal2 = pair_balance("{","}");
15
16 negative = bal1<0 or bal2<0;
17 had_neg = aggregate(select_all,
18                 indicator(negative))>0;
19 select_last = select(indices,length-1,==);
20 end_0 = aggregate(select_last,
21              bal1==0 and bal2==0);
22
23 shuffle_dyck2 = end_0 and not had_neg;
```

Figure 3: RASP program for the task shuffle-dyck-2 (balance 2 parenthesis pairs, independently of each other), capturing a higher level representation of the hand-crafted transformer presented by Bhattamishra et al. (2020).

Example: Shuffle-Dyck in RASP As an example of the kind of tasks that are natural to encode using RASP, consider the Shuffle-Dyck language, in which multiple parentheses types must be balanced but do not have to satisfy any order with relation to each other. (For example, `"([)]"` is considered balanced). In their work on transformer expressiveness, Bhattamishra et al. (2020) present a hand-crafted transformer for this language, including the details of which dimension represents which partial computation. RASP can concisely describe the same solution, showing the high-level operations while abstracting away the details of their arrangement into an actual transformer architecture.

We present this solution in Figure 3: the code compiles to a transformer architecture using 2 layers and a total of 3 heads, exactly as in the construction of Bhattamishra et al.. These numbers are inferred by the RASP compiler: the programmer does not have to think about such details.

A pair of parentheses is balanced in a sequence if their running balance is never negative, and additionally is equal to exactly 0 at the final input token. Lines 13–23 check this definition: lines 13 and 14 use `pair_balance` to compute the running balances of each parenthesis pair, and 17 checks whether these balances were negative anywhere in the sequence. The snippet in 21 (`bal1==0 and bal2==0`) creates an s-op checking at each location whether both pairs are balanced, with the aggregation of line 20 loading the value of this s-op from the last position. From there, a boolean composition of `end_0` and `had_neg` defines shuffle-dyck-2.

Compilation and Abstraction The high-level operations in RASP can be compiled down to execute on a transformer: for example, the code presented in Figure 1 compiles to a two-layer, 3-head (total) architecture, whose attention patterns when applied to the input sequence `"§aaabbccdef"` are presented in Figure 1(b). (The full compiled computation flow for this program—showing how its component s-ops interact—is presented in Appendix B).

RASP abstracts away low-level operations into simple primitives, allowing a programmer to explore the full potential of a transformer without getting bogged down in the details of how these are realized in practice. At the same time, RASP enforces the information-flow constraints of transformers, preventing anyone from writing a program more powerful than they can express. One example of this is the lack of input-dependent loops in the s-ops, reflecting the fact that transformers cannot arbitrarily repeat operations[2]. Another is in the selectors: for each two positions, the decision whether one selects ('attends to') the other is pairwise.

We find RASP a natural tool for conveying transformer solutions to given tasks. It is modular and compositional, allowing us to focus on arbitrarily high-level computations when writing programs. Of course, we are restricted to tasks for which a human *can* encode a solution: we do not expect any researcher to implement, e.g., a strong language model or machine-translation system in RASP—these are not realizable in any programming language. Rather, we focus on programs that convey concepts that people can encode in "traditional" programming languages, and the way they relate to the expressive power of the transformer.

In Section 5, we will show empirically that RASP solutions can indeed translate to real transformers. One example is given in Figure 1: having written a RASP program (left) for the double-histograms task, we analyse it to obtain the number of layers and heads needed for a transformer to mimic our solution, and then train a transformer with supervision of both its outputs and its attention patterns to obtain a neural version of our solution (right). We find that the transformer can accurately learn the target attention patterns and use them to reach a high accuracy on the target task.

[2] Though work exploring such transformer variants exists: Dehghani et al. (2019) devise a transformer architecture with a control unit, which can repeat its sublayers arbitrarily many times.

3. The RASP language

RASP contains a small set of primitives and operations built around the core task of manipulating sequence processing functions referred to as *s-ops (sequence operators)*, functions that take in an input sequence and return an output sequence of the same length. Excluding some atomic values, and the convenience of lists and dictionaries, *everything* in RASP is a function. Hence, to simplify presentation, we often demonstrate RASP values with one or more input-output pairs: for example, `identity("hi")="hi"`[3].

RASP has a small set of built-in s-ops, and the goal of programming in RASP is to compose these into a final s-op computing the target task. For these compositions, the functions `select` (creating selection matrices called *selectors*), `aggregate` (collapsing selectors and s-ops into a new s-ops), and `selector_width` (creating an s-op from a selector) are provided, along with several elementwise operators reflecting the feed-forward sublayers of a transformer. As noted in Section 2, while all s-ops and selectors are in fact functions, we will prefer to talk in terms of the sequences and matrices that they create. Constant values in RASP (e.g., 2, T, `h`) are treated as s-ops with a single value broadcast at all positions, and all symbolic values are assumed to have an underlying numerical representation which is the value being manipulated in practice.

The built-in s-ops The simplest s-op is the identity, given in RASP under the name `tokens`: `tokens("hi")="hi"`. The other built-in s-ops are `indices` and `length`, processing input sequences as their names suggest: `indices("hi")=[0,1]`, and `length("hi")=[2,2]`.

s-ops can be combined with constants (numbers, booleans, or tokens) or each other to create new s-ops, in either an elementwise or more complicated fashion.

Elementwise combination of s-ops is done by the common operators for the values they contain, for example: `(indices+1)("hi")=[1,2]`, and `((indices+1)==length)("hi")=[F,T]`. This includes also a ternary operator: `(tokens if (indices%2==0) else "-")("hello")="h-l-o"`. When the condition of the operator is an s-op itself, the result is an s-op that is dependent on all 3 of the terms in the operator creating it.

Select and Aggregate operations are used to combine information from different sequence positions. A selector takes two lists, representing *keys* and *queries* respectively, and a predicate p, and computes from these a selection matrix describing for each key, query pair (k, q) whether the condition $p(k, q)$ holds.

[3] We use strings as shorthand for a sequence of characters.

For example,

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

An aggregate operation takes one selection matrix and one list, and averages for each row of the matrix the values of the list in its selected columns. For example,

$$agg(\begin{bmatrix} \mathbf{T} & F & F \\ \mathbf{T} & \mathbf{T} & F \\ \mathbf{T} & \mathbf{T} & \mathbf{T} \end{bmatrix}, [10,20,30]) = [10,15,20]$$

Intuitively, a select-aggregate pair can be thought of as a two-dimensional map-reduce operation. The selector can be viewed as performing filtering, and aggregate as performing a reduce operation over the filtered elements (see Figure 2).

In RASP, the selection operation is provided through the function `select`, which takes two s-ops `k` and `q` and a comparison operator $\circ$ and returns the composition of $\text{sel}(\cdot,\cdot,\circ)$ with `k` and `q`, with this sequence-to-matrix function referred to as a *selector*. For example: `a=select(indices,indices,<)` is a selector, and `a("hey")=[[F,F,F],[T,F,F],[T,T,F]]`. Similarly, the aggregation operation is provided through `aggregate`, which takes one selector and one s-op and returns the composition of agg with these. For example: `aggregate(a,indices+1)("hey")=[0,1,1.5]`.[4]

Simple select-aggregate examples To create the s-op that reverses any input sequence, we build a selector that requests for each query position the token at the opposite end of the sequence, and then aggregate that selector with the original input tokens: `flip=select(indices,length-indices-1,==)`, and `reverse=aggregate(flip,tokens)`. For example:

$$\texttt{flip("hey")} = \begin{bmatrix} F & F & \mathbf{T} \\ F & \mathbf{T} & F \\ \mathbf{T} & F & F \end{bmatrix}$$

$$\texttt{reverse("hey")} = \texttt{"yeh"}$$

To compute the fraction of appearances of the token `"a"` in our input, we build a selector that gathers information from *all* input positions, and then aggregate it with a sequence broadcasting 1 wherever the input token is `"a"`, and 0 everywhere else. This is expressed as `select_all=select(1,1,==)`, and then `frac_as = aggregate(select_all,1 if tokens=="a" else 0)`.

[4]For convenience and efficiency, when averaging the filtered values in an aggregation, for every position where only a single value has been selected, RASP passes that value directly to the output without attempting to 'average' it. This saves the programmer from unnecessary conversion into and out of numerical representations when making simple transfers of tokens between locations: for example, using the selector `load1=select(indices,1,==)`, we may directly create the s-op `aggregate(load1,tokens)("hey")="eee"`. Additionally, in positions when no values are selected, the aggregation simply returns a default value for the output (in Figure 2, we see this with default value 0), this value may be set as one of the inputs to the `aggregate` function.

Selector manipulations Selectors can be combined elementwise using boolean logic. For example, for the same `load1` and `flip` from above:

$$\texttt{(load1 or flip)("hey")} = \begin{bmatrix} F & \mathbf{T} & \mathbf{T} \\ F & \mathbf{T} & F \\ \mathbf{T} & \mathbf{T} & F \end{bmatrix}$$

selector width The final operation in RASP is the powerful `selector_width`, which takes as input a single selector and returns a new s-op that computes, for each output position, the number of input values which that selector has chosen for it. This is best understood by example: using the selector `same_token=select(tokens,tokens,==)` that filters for each query position the keys with the same token as its own, we can compute its width to obtain a histogram of our input sequence: `selector_width(same_token)("hello")=[1,1,2,2,1]`.

Additional operations: While the above operations are together sufficient to represent any RASP program, RASP further provides a library of primitives for common operations, such as `in` – either of a value within a sequence: `("i" in tokens)("hi")=[T,T]`, or of each value in a sequence within some static list: `tokens in ["a","b","c"])("hat")=[F,T,F]`. RASP also provides functions such as `count`, or `sort`.

3.1. Relation to a Transformer

We discuss how the RASP operations compile to describe the information flow of a transformer architecture, suggesting how many heads and layers are needed to solve a task.

The built-in s-ops `indices` and `tokens` reflect the initial input embeddings of a transformer, while `length` is computed in RASP: `length= 1/aggregate(select_all,indicator(indices==0))`, where `select_all=select(1,1,==)`.

Elementwise Operations reflect the feed-forward sub-layers of a transformer. These have overall not been restricted in any meaningful way: as famously shown by Hornik et al. (1989), MLPs such as those present in the feed-forward transformer sub-layers can approximate with arbitrary accuracy any borel-measurable function, provided sufficiently large input and hidden dimensions.

Selection and Aggregation Selectors translate to attention matrices, defining for each input the selection (attention) pattern used to mix the input values into a new output through weighted averages, and aggregation reflects this final averaging operation. The uniform weights dictated by our selectors reflect an attention pattern in which 'unselected' pairs are all given strongly negative scores, while the selected pairs all have higher, similar, scores. Such attention patterns are supported by the findings of (Merrill et al., 2020a).

Decoupling selection and aggregation in RASP allows `selectors` to be reused in multiple aggregations, abstracting away the fact that these may actually require separate attention heads in the compiled architecture. Making selectors first class citizens also enables functions such as `selector_width`, which take selectors as parameters.

Additional abstractions All other operations, including the powerful `selector_width` operation, are implemented in terms of the above primitives. `selector_width` in particular can be implemented such that it compiles to either one or two selectors, depending on whether or not one can assume a beginning-of-sequence token is added to the input sequence. Its implementation is given in Appendix B.

Compilation Converting an s-op to a transformer architecture is as simple as tracing its computation flow out from the base s-ops. Each aggregation is an attention head, which must be placed at a layer later than all of its inputs. Elementwise operations are feedforward operations, and sit in the earliest layer containing all of their dependencies. Some optimisations are possible: for example, aggregations performed at the same layer with the same selector can be merged into the same attention head. A "full" compilation—to concrete transformer weights—requires to e.g. derive MLP weights for the elementwise operations, and is beyond the scope of this work. RASP provides a method to visualize the compiled computation flow of any s-op and input pair: the flows in Figs 4 and 5 were rendered using `draw(reverse,"abcde")` and `draw(hist,"§aabbaabb")`.

4. Implications and insights

Restricted-Attention Transformers Multiple works propose restricting the attention mechanism to create more efficient transformers, reducing the time complexity of each layer from O(n^2) to O($nlog(n)$) or even O(n) with respect to the input sequence length n (see Tay et al. (2020) for a survey of such approaches). Several of these do so using *sparse attention*, in which the attention is masked using different patterns to reduce the number of locations that can interact ((Child et al., 2019; Beltagy et al., 2020; Ainslie et al., 2020; Zaheer et al., 2020; Roy et al., 2021)).

Considering such transformer variants in terms of RASP allows us to reason about the computations they can and cannot perform. In particular, these variants of transformers all impose restrictions on the selectors, permanently forcing some of the n^2 index pairs in every selector to `False`. But does this necessarily weaken these transformers?

In Appendix B we present a sorting algorithm in RASP, applicable to input sequences with arbitrary length and alphabet size[5]. This problem is known to require at $\Omega(n\log(n))$ operations in the input length n—implying that a standard transformer can take full advantage of $\Omega(n\log(n))$ of the n^2 operations it performs in every attention head. It follows from this that all variants restricting their attention to $o(n\log(n))$ operations incur a real loss in expressive power.

Sandwich Transformers Recently, Press et al. (2020) showed that reordering the attention and feed-forward sublayers of a transformer affects its ability to learn language modeling tasks. In particular, they showed that: 1. pushing feed-forward sublayers towards the bottom of a transformer weakened it; and 2. pushing attention sublayers to the bottom and feed-forward sublayers to the top strengthened it, provided there was still some interleaving in the middle.

The base operations of RASP help us understand the observations of Press et al.. Any arrangement of a transformer's sublayers into a fixed architecture imposes a restriction on the number and order of RASP operations that can be chained in a program compilable to that architecture. For example, an architecture in which all feed-forward sublayers appear before the attention sublayers, imposes that no elementwise operations may be applied to the result of any aggregation.

In RASP, there is little value to repeated elementwise operations before the first `aggregate`: each position has only its initial input, and cannot generate new information. This explains the first observation of Press et al.. In contrast, an architecture beginning with several attention sublayers—i.e., multiple `select-aggregate` pairs—will be able to gather a large amount of information into each position early in the computation, even if only by simple rules[6]. More complicated gathering rules can later be realised by applying elementwise operations to aggregated information before generating new selectors, explaining the second observation.

Recognising Dyck-k Languages The Dyck-k languages—the languages of sequences of correctly balanced parentheses, with k parenthesis types—have been heavily used in considering the expressive power of RNNs (Sennhauser & Berwick, 2018; Skachkova et al., 2018; Bernardy, 2018; Merrill, 2019; Hewitt et al., 2020).

Such investigations motivate similar questions for transformers, and several works approach the task. Hahn (2020) proves that transformer-encoders with hard attention cannot recognise Dyck-2. Bhattamishra et al. (2020) and Yao et al. (2021) provide transformer-encoder constructions

[5] Of course, realizing this solution in real transformers requires sufficiently stable word and positional embeddings—a practical limitation that applies to all transformer variants.

[6] While the attention sublayer of a transformer does do some local manipulations on its input to create the candidate output vectors, it does not contain the powerful MLP with hidden layer as is present in the feed-forward sublayer.

for recognizing simplified variants of Dyck-k, though the simplifications are such that no conclusion can be drawn for unbounded depth Dyck-k with $k > 1$. Optimistically, Ebrahimi et al. (2020) train a transformer-encoder with causal attention masking to process Dyck-k languages with reasonable accuracy for several $k > 1$, finding that it learns a stack-like behaviour to complete the task.

We consider Dyck-k using RASP, specifically defining Dyck-k-PTF as the task of classifying for every prefix of a sequence whether it is legal, but not yet balanced (**P**), balanced (**T**), or illegal (**F**). We show that RASP can solve this task in a fixed number of heads and layers for *any* k, presenting our solution in Appendix B[7].

Symbolic Reasoning in Transformers Clark et al. (2020) show that transformers are able to emulate symbolic reasoning: they train a transformer which, given the facts "Ben is a bird" and "birds can fly", correctly validates that "Ben can fly". Moreover, they show that transformers are able to perform several logical 'steps': given also the fact that only winged animals can fly, their transformer confirms that Ben has wings. This finding however does not shed any light on *how* the transformer is achieving such a feat.

RASP empowers us to approach the problem on a high level. We write a RASP program for the related but simplified problem of containment and inference over sets of elements, sets, and logical symbols, in which the example is written as `b∈B, x∈B→x∈F, b∈F?` (implementation available in our repository). The main idea is to store at the position of each set symbol the elements contained and not contained in that set, and at each element symbol the sets it is and is not contained in. Logical inferences are computed by passing information between symbols in the same 'fact', and propagated through pairs of identical set or element symbols, which share their stored information.

Use of Separator Tokens Clark et al. (2019) observe that many attention heads in BERT (Devlin et al., 2019) (sometimes) focus on separator tokens, speculating that these are used for "no-ops" in the computation. (Ebrahimi et al., 2020) find that transformers more successfully learn Dyck-k languages when the input is additionally provided with a beginning-of-sequence (BOS) token, with the trained models treating it as a base in their stack when there are no open parentheses. Our RASP programs suggest an additional role that such separators may be playing: by providing a fixed signal from a 'neutral' position, separators facilitate conditioned counting in transformers, that use the diffusion of the signal to compute how many positions a head was attending to. Without such neutral positions, counting requires an additional head, such that an agreed-upon position may artificially be treated as neutral in one head and then independently accounted for in the other.

[7] We note that RASP does not suggest the embedding width needed to encode this solution in an actual transformer.

A simple example of this is seen in Figure 5. There, `selector_width` is applied with a BOS token, creating in the process an attention pattern that focuses on the first input position (the BOS location) from all query positions, in addition to the actual positions selected by `select(tokens,tokens,==)`. A full description of `selector_width` is given in Appendix B.

5. Experiments

We evaluate the relation of RASP to transformers on three fronts: 1. its ability to upper bound the number of heads and layers required to solve a task, 2. the tightness of that bound, 3. its feasibility in a transformer, i.e., whether a sufficiently large transformer can encode a given RASP solution., training several transformers. We relegate the exact details of the transformers and their training to Appendix A.

For this section, we consider the following tasks:

1. Reverse, e.g.: `reverse("abc")="cba"`.
2. Histograms, with a unique beginning-of-sequence (BOS) token § (e.g., `hist_bos("§aba")=[§,2,1,2]`) and without it (e.g., `hist_nobos("aba")=[2,1,2]`).
3. Double-Histograms, with BOS: for each token, the number of unique tokens with same histogram value as itself. E.g.: `hist2("§abbc")=[§,2,1,1,2]`.
4. Sort, with BOS: ordering the input tokens lexicographically. e.g.: `sort("§cba")="§abc"`.
5. Most-Freq, with BOS: returning the unique input tokens in order of decreasing frequency, with original position as a tie-breaker and the BOS token for padding. E.g.: `most_freq("§abbccddd")="§dbca§§§§"`.
6. Dyck-i PTF, for $i = 1, 2$: the task of returning, at each output position, whether the input prefix up to and including that position is a legal Dyck-i sequence (`T`), and if not, whether it can (`P`) or cannot (`F`) be continued into a legal Dyck-i sequence. E.g: `Dyck1_ptf("()())")="PTPTF"`.

We refer to double-histogram as 2-hist, and to each Dyck-i PTF problem simply as Dyck-i. The full RASP programs for these tasks, and the computation flows they compile down to, are presented in Appendix B. The size of the transformer architecture each task compiles to is presented in Table 1.

Upper bounding the difficulty of a task Given a RASP program for a task, e.g. double-histogram as described in Figure 1, we can compile it down to a transformer architec-

[8] The actual optimal solution for Dyck-2 PTF cannot be realised in RASP as is, as it requires the addition of a select-*best* operator to the language—reflecting the power afforded by softmax in the transformer's self-attention. In this paper, we always refer to our analysis of Dyck-2 with respect to this additional operation.

Language	Layers	Heads	Test Acc.	Attn. Matches?
Reverse	2	1	99.99%	✓✗
Hist BOS	1	1	100%	✓
Hist no BOS	1	2	99.97%	✓✗
Double Hist	2	2	99.58%	✓✗
Sort	2	1	99.96%	✗
Most Freq	3	2	95.99%	✗
Dyck-1 PTF	2	1	99.67%	✓✗
Dyck-2 PTF [8]	3	1	99.85%	✗

Table 1: Does a RASP program correctly upper bound the number of heads and layers needed for a transformer to solve a task? In the left columns, we show the compilation size of our RASP programs for each considered task, and in the right columns we show the best (of 4) accuracies of transformers trained on these same tasks, and evaluate whether their attention mechanisms appear to match (using a ✓✗ for partially similar patterns: see Figure 4 for an example). For RASP programs compiling to varying number of heads per layer, we report the maximum of these.

ture, effectively predicting the maximum number of layers and layer width (number of heads in a layer) needed to solve that task in a transformer. To evaluate whether this bound is truly sufficient for the transformer, we train 4 transformers of the prescribed sizes on each of the tasks.

We report the accuracy of the best trained transformer for each task in Table 1. Most of these transformers reached accuracies of 99.5% and over, suggesting that the upper bounds obtained by our programs are indeed sufficient for solving these tasks in transformers. For some of the tasks, we even find that the RASP program is the same as or very similar to the 'natural' solution found by the trained transformer. In particular, Figures 4 and 5 show a strong similarity between the compiled and learned attention patterns for the tasks Reverse and Histogram-BOS, though the transformer trained on Reverse appears to have learned a different mechanism for computing `length` than that given in RASP.

Tightness of the bound We evaluate the tightness of our RASP programs by training smaller transformers than those predicted by our compilation, and observing the drop-off in test accuracy. Specifically, we repeat our above experiments, but this time we also train each task on up to 4 different sizes. In particular, denoting L, H the number of layers and heads predicted by our compiled RASP programs, we train for each task transformers with sizes (L, H), $(L-1, H)$, $(L, H-1)$, and $(L-1, 2H)$ (where possible) [9].

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

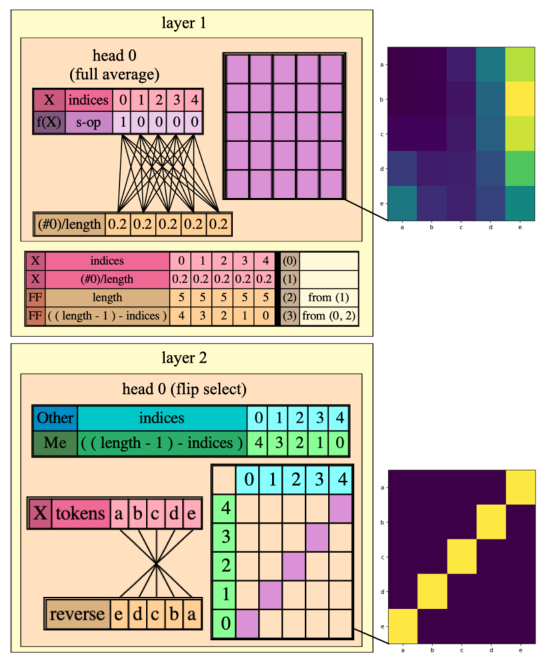

Figure 4: Top: RASP code for computing reverse (e.g., `reverse("abc")="cba"`). Below, its compilation to a transformer architecture (left, obtained through `draw(reverse,"abcde")` in the RASP REPL), and the attention heatmaps of a transformer trained on the same task (right), both visualised on the same input. Visually, the attention head in the second layer of this transformer corresponds perfectly to the behavior of the `flip` selector described in the program. The head in the first layer, however, appears to have learned a different solution from our own: instead of focusing uniformly on the entire sequence (as is done in the computation of `length` in RASP), this head shows a preference for the last position in the sequence.

We report the average test accuracy reached by each of these architectures in Table 2. For most of the tasks, the results show a clear drop in accuracy as the number of heads or layers is reduced below that obtained by our compiled RASP solutions for the same tasks—several of these reduced transformers fail completely to learn their target languages.

The main exception to this is sort, which appears unaffected by the removal of one layer, and even achieves its best results in this case. Drawing the attention pattern for the single-layer sort transformers reveals relatively uniform attention patterns. It appears that the transformer has learned to take advantage of the bounded input alphabet size, effectively

[9] The transformers of size $(L-1, 2H)$ are used to validate that any drop in accuracy is indeed due to the reduction in number of layers, as opposed to the reduction in total heads that this entails. However, doubling H means the embedding dimension will be divided over twice as many heads. To counteract any negative effect this may have, we also double the embedding dimension for these transformers.

```
same_tok = select(tokens, tokens, ==);
hist = selector_width(same_tok,
            assume_bos = True);
```

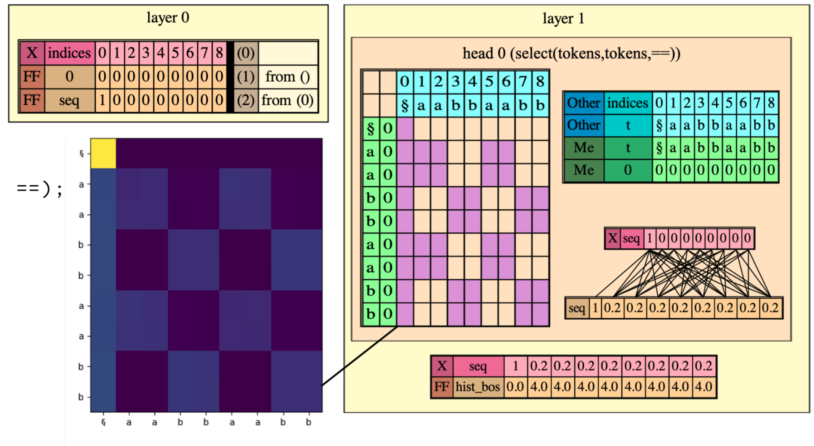

Figure 5: The RASP program for computing with-BOS histograms (left), alongside its compilation to a transformer architecture (cream boxes) and the attention head (center bottom) of a transformer trained on the same task, without attention supervision. The compiled architecture and the trained head are both presented on the same input sequence, `"§aabbaabb"`. The transformer architecture was generated in the RASP REPL using `draw(hist,"§aabbaabb")`.

Language	RASP	Average test accuracy (%) with...			
	L, H	L, H	$H-1$	$L-1$	$L-1, 2H$
Reverse	$2, 1$	**99.9**	-	23.1	41.2
Hist	$1, 2$	**99.9**	91.9	-	-
2-Hist	$2, 2$	**99.0**	73.5	40.5	83.5
Sort	$2, 1$	99.8	-	99.0	**99.9**
Most Freq	$3, 2$	**93.9**	92.1	84.0	90.2
Dyck-1	$2, 1$	**99.3**	-	96.9	96.4
Dyck-2	$3, 1$	**99.7**	-	98.8	94.1

Table 2: Accuracy dropoff in transformers when reducing their number of heads and layers relative to the compiled RASP solutions for the same tasks. The transformers trained on the size predicted by RASP have very high accuracy, and in most cases there is a clear drop as that size is reduced. Cases creating an impossible architecture (H or L zero) are marked with -. Histogram with BOS uses only 1 layer and 1 head, and so is not included. As in Table 1, Dyck-2 is considered with the addition of `select_best` to RASP.

implementing bucket sort for its task. This is because a single full-attention head is sufficient to compute for every token its total appearances in the input, from which the correct output can be computed locally at every position.

Feasibility of a RASP program We verify that a given RASP program can indeed be represented in a transformer. For this, we return to the tougher tasks above, and this time train the transformer with an additional loss component encouraging it to learn the attention patterns created in our compiled solution (i.e., we supervise the attention patterns in addition to the target output). In particular, we consider the tasks double-histogram, sort, and most-freq, all with the assumption of a BOS token in the input. After training each transformer for 250 epochs with both target and attention supervision, they all obtain high test accuracies on the task (99+%), and appear to encode attention patterns similar to those compiled from our solutions. We present the obtained patterns for double-histogram, alongside the compiled RASP solution, in Figure 1. We present its full computation flow, as well as the learned attention patterns and full flow of sort and most-freq, in Appendix A.

6. Conclusions

We abstract the computation model of the transformer-encoder as a simple sequence processing language, RASP, that captures the unique constraints on information flow present in a transformer. Considering computation problems and their implementation in RASP allows us to "think like a transformer" while abstracting away the technical details of a neural network in favor of symbolic programs. We can analyze any RASP program to infer the minimum number of layers and maximum number of heads required to realise it in a transformer. We show several examples of programs written in the RASP language, showing how operations can be implemented by a transformer, and train several transformers on these tasks, finding that RASP helps predict the number of transformer heads and layers needed to solve them. Additionally, we use RASP to shed light on an empirical observation over transformer variants, and find concrete limitations for some "efficient transformers".

Acknowledgments We thank Uri Alon, Omri Gilad, and the reviewers for their constructive comments. This project received funding from European Research Council (ERC) under Europoean Union's Horizon 2020 research and innovation programme, agreement No. 802774 (iEXTRACT).

References

Ainslie, J., Ontañón, S., Alberti, C., Pham, P., Ravula, A., and Sanghai, S. ETC: encoding long and structured data in transformers. *CoRR*, abs/2004.08483, 2020. URL https://arxiv.org/abs/2004.08483.

Ayache, S., Eyraud, R., and Goudian, N. Explaining black boxes on sequential data using weighted automata. In *Proceedings of the 14th International Conference on Grammatical Inference, ICGI*, 2018. URL http://proceedings.mlr.press/v93/ayache19a.html.

Bahdanau, D., Cho, K., and Bengio, Y. Neural machine translation by jointly learning to align and translate. In Bengio, Y. and LeCun, Y. (eds.), *3rd International Conference on Learning Representations, ICLR*, 2015. URL http://arxiv.org/abs/1409.0473.

Beltagy, I., Peters, M. E., and Cohan, A. Longformer: The long-document transformer. *CoRR*, abs/2004.05150, 2020. URL https://arxiv.org/abs/2004.05150.

Bernardy, J.-P. Can recurrent neural networks learn nested recursion? In *Linguistic Issues in Language Technology, Volume 16*. CSLI Publications, 2018. URL https://www.aclweb.org/anthology/2018.lilt-16.1.

Bhattamishra, S., Ahuja, K., and Goyal, N. On the ability and limitations of transformers to recognize formal languages. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, EMNLP*, 2020. URL https://doi.org/10.18653/v1/2020.emnlp-main.576.

Child, R., Gray, S., Radford, A., and Sutskever, I. Generating long sequences with sparse transformers. *CoRR*, abs/1904.10509, 2019. URL http://arxiv.org/abs/1904.10509.

Clark, K., Khandelwal, U., Levy, O., and Manning, C. D. What does BERT look at? an analysis of bert's attention. *CoRR*, abs/1906.04341, 2019. URL http://arxiv.org/abs/1906.04341.

Clark, P., Tafjord, O., and Richardson, K. Transformers as soft reasoners over language. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI*, 2020. URL https://doi.org/10.24963/ijcai.2020/537.

Cleeremans, A., Servan-Schreiber, D., and McClelland, J. L. Finite state automata and simple recurrent networks. *Neural Comput.*, 1(3):372–381, 1989. URL https://doi.org/10.1162/neco.1989.1.3.372.

Dehghani, M., Gouws, S., Vinyals, O., Uszkoreit, J., and Kaiser, L. Universal transformers. In *7th International Conference on Learning Representations, ICLR*, 2019. URL https://openreview.net/forum?id=HyzdRiR9Y7.

Devlin, J., Chang, M., Lee, K., and Toutanova, K. BERT: pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT*, 2019. URL https://doi.org/10.18653/v1/n19-1423.

Ebrahimi, J., Gelda, D., and Zhang, W. How can self-attention networks recognize Dyck-n languages? In *Findings of the Association for Computational Linguistics: EMNLP*. Association for Computational Linguistics, 2020. doi: 10.18653/v1/2020.findings-emnlp.384. URL https://www.aclweb.org/anthology/2020.findings-emnlp.384.

Hahn, M. Theoretical limitations of self-attention in neural sequence models. *Trans. Assoc. Comput. Linguistics*, 2020. URL https://transacl.org/ojs/index.php/tacl/article/view/1815.

Hewitt, J., Hahn, M., Ganguli, S., Liang, P., and Manning, C. D. Rnns can generate bounded hierarchical languages with optimal memory. In Webber, B., Cohn, T., He, Y., and Liu, Y. (eds.), *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing, EMNLP*, 2020. URL https://doi.org/10.18653/v1/2020.emnlp-main.156.

Hornik, K., Stinchcombe, M. B., and White, H. Multilayer feedforward networks are universal approximators. *Neural Networks*, 2(5):359–366, 1989. URL https://doi.org/10.1016/0893-6080(89)90020-8.

Joulin, A. and Mikolov, T. Inferring algorithmic patterns with stack-augmented recurrent nets. In *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems, NeurIPS*, 2015. URL https://arxiv.org/abs/1503.01007.

Luong, T., Pham, H., and Manning, C. D. Effective approaches to attention-based neural machine translation. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing, EMNLP*, 2015. URL https://doi.org/10.18653/v1/d15-1166.

Merrill, W. Sequential neural networks as automata. *CoRR*, 2019. URL http://arxiv.org/abs/1906.01615.

Merrill, W., Ramanujan, V., Goldberg, Y., Schwartz, R., and Smith, N. A. Parameter norm growth during training of transformers. *CoRR*, 2020a. URL https://arxiv.org/abs/2010.09697.

Merrill, W., Weiss, G., Goldberg, Y., Schwartz, R., Smith, N. A., and Yahav, E. A formal hierarchy of RNN architectures. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, 2020b. URL https://www.aclweb.org/anthology/2020.acl-main.43.

Omlin, C. W. and Giles, C. L. Extraction of rules from discrete-time recurrent neural networks. *Neural Networks*, 9(1):41–52, 1996. URL https://doi.org/10.1016/0893-6080(95)00086-0.

Pérez, J., Barceló, P., and Marinkovic, J. Attention is turing-complete. *Journal of Machine Learning Research*, 22, 2021. URL http://jmlr.org/papers/v22/20-302.html.

Press, O., Smith, N. A., and Levy, O. Improving transformer models by reordering their sublayers. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics, ACL*, 2020. URL https://www.aclweb.org/anthology/2020.acl-main.270/.

Rabusseau, G., Li, T., and Precup, D. Connecting weighted automata and recurrent neural networks through spectral learning. In *The 22nd International Conference on Artificial Intelligence and Statistics, AISTATS*, volume 89 of *Proceedings of Machine Learning Research*, 2019. URL http://proceedings.mlr.press/v89/rabusseau19a.html.

Roy, A., Saffar, M., Vaswani, A., and Grangier, D. Efficient content-based sparse attention with routing transformers. *Trans. Assoc. Comput. Linguistics*, 9, 2021. URL https://transacl.org/ojs/index.php/tacl/article/view/2405.

Sennhauser, L. and Berwick, R. C. Evaluating the ability of lstms to learn context-free grammars. In *Proceedings of the Workshop: Analyzing and Interpreting Neural Networks for NLP, BlackboxNLP@EMNLP*, pp. 115–124, 2018. URL https://doi.org/10.18653/v1/w18-5414.

Skachkova, N., Trost, T., and Klakow, D. Closing brackets with recurrent neural networks. In *Proceedings of the Workshop: Analyzing and Interpreting Neural Networks for NLP, BlackboxNLP@EMNLP*, pp. 232–239, 2018. URL https://doi.org/10.18653/v1/w18-5425.

Tay, Y., Dehghani, M., Bahri, D., and Metzler, D. Efficient transformers: A survey. *CoRR*, abs/2009.06732, 2020. URL https://arxiv.org/abs/2009.06732.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems, NeurIPS*, 2017. URL https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html.

Weiss, G., Goldberg, Y., and Yahav, E. On the practical computational power of finite precision rnns for language recognition. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL*, 2018a. URL https://aclanthology.info/papers/P18-2117/p18-2117.

Weiss, G., Goldberg, Y., and Yahav, E. Extracting automata from recurrent neural networks using queries and counterexamples. In *Proceedings of the 35th International Conference on Machine Learning, ICML*, 2018b. URL http://proceedings.mlr.press/v80/weiss18a.html.

Yao, S., Peng, B., Papadimitriou, C., and Narasimhan, K. Self-attention networks can process bounded hierarchical languages. *CoRR*, 2021. URL https://arxiv.org/abs/2105.11115.

Yun, C., Bhojanapalli, S., Rawat, A. S., Reddi, S. J., and Kumar, S. Are transformers universal approximators of sequence-to-sequence functions? In *8th International Conference on Learning Representations, ICLR*, 2020. URL https://openreview.net/forum?id=ByxRM0Ntvr.

Zaheer, M., Guruganesh, G., Dubey, K. A., Ainslie, J., Alberti, C., Ontañón, S., Pham, P., Ravula, A., Wang, Q., Yang, L., and Ahmed, A. Big bird: Transformers for longer sequences. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems, NeurIPS*, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/c8512d142a2d849725f31a9a7a361ab9-Abstract.html.