ADAPTIVE CONTROL FLOW IN TRANSFORMERS IMPROVES SYSTEMATIC GENERALIZATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Despite successes across a broad range of applications, Transformers have limited capability in systematic generalization. The situation is especially frustrating in the case of algorithmic tasks, where they often fail to find intuitive solutions that can be simply expressed in terms of attention patterns. In the end, it is often all about routing the right information to the right node/operation at the right time in the grid represented by Transformer columns. To facilitate the learning of useful control flow, we propose two modifications to the Transformer architecture, copy gate and geometric attention. Our novel Transformer Control Flow (TCF) achieves 100% length generalization accuracy on the classic compositional table lookup task, as well as near-perfect accuracy on the simple arithmetic task and a new variant of ListOps testing for computational depth generalization. TCF's attention and gating patterns tend to be interpretable.

1 INTRODUCTION

Neural networks (NNs) typically do not generalize to a systematically different test set, even when they can easily solve the training set. Examples of systematic generalization (Fodor et al., 1988) include generalization to sequences longer than those seen during training – productivity, and algorithmic combinations of previously learned rules – systematicity. Despite recent efforts (Bahdanau et al., 2019; Korrel et al., 2019; Lake, 2019; Li et al., 2019; Russin et al., 2019; Csordás et al., 2021), systematic generalization generally remains unsolved (Fodor & McLaughlin, 1990; Lake & Baroni, 2018; Liska et al., 2018; Greff et al., 2020; Hupkes et al., 2020). On some datasets, the best performing models are neuro-symbolic hybrids (Chen et al., 2020; Liu et al., 2020) using task-specific symbolic functions. However, their applicability to other datasets remains limited (Furrer et al., 2020; Shaw et al., 2020). A big question is: which type of architectural inductive bias encourages the training process to select "good" solutions which systematically generalize?

The popular Transformer (Vaswani et al., 2017) architecture also suffers from such problems (Chaabouni et al., 2021; Csordás et al., 2021; Ontañón et al., 2021). Frustratingly, Transformers fail to generalize in many algorithmic tasks (e.g. Liska et al. (2018); Dubois et al. (2020)), even those that have intuitive solutions which can be simply expressed in terms of Transformer attention patterns. Indeed, given an input sequence of length N and a Transformer encoder of depth T, solving an algorithmic task is often all about routing the right information to the right node/operation at the right time in the T-by-N grid represented by Transformer columns. Effectively the task is to learn to draw an *adaptive control flow* on the canvas of Transformer columns. In fact, recent work by Weiss et al. (2021) introduced a programming language called RASP which is specifically designed to express solutions to sequence processing problems, and which has a direct equivalent to the operations in Transformers. However, this work also shows that making Transformers learn a solution expressed in RASP is only possible with intermediate supervision of attention patterns. In some cases, even such supervision fails (Weiss et al., 2021). In general, Transformers fail to find easily interpretable and/or symbolic solutions to algorithmic tasks. We conversely hypothesize that attention-based NNs able to find intuitive solutions (achieving interpretable attention patterns) can improve systematic generalization.

Here we argue that regular Transformers lack some basic ingredients for learning such "intuitive" solutions to algorithmic problems. As a remedy, we propose simple architectural modifications to help them learn data routing. As a first step towards validating our model, we focus on the popular length generalization task of compositional table lookup (CTL; Liska et al. (2018); Hupkes et al. (2019);

Dubois et al. (2020)), as well as two more complex tasks: a simple arithmetic task and a variant of ListOps (Nangia & Bowman, 2018) which are designed to test NNs' ability for compositional generalization. Our novel Transformer Control Flow (TCF) achieves 100% generalization accuracy (never reported before; Dubois et al. (2020)) on the CTL task, and obtains nearly perfect accuracy on both the proposed simple arithmetic and ListOps tasks. We show that the attention and gating patterns of TCF tend to be interpretable as plausible control flows.

2 Improving Transformers for Learning Adaptive Control Flow

We argue that the following components are needed to build Transformers capable of learning adaptive control flow. **First**, composing known operations in an arbitrary order requires that all operations are available at every computational step. This can be easily achieved by sharing the weights of the layers, as is done in Universal Transformers (Dehghani et al., 2019). **Second**, the network should be sufficiently deep, at least as deep as the deepest data dependency in the computational graph built from elementary operations (e.g., in case of a parse tree, this is the depth of the tree). Otherwise, multiple operations would be fused into a single layer and hinder natural and elegant compositions. **Third**, inputs in some columns should be kept unchanged until the right point in time when they can be processed in a later step/layer. The regular Transformer lacks a mechanism for skipping the whole transformation step by simply copying the input to the next step/layer. We propose a special gating function, *copy gate*, to implement such a mechanism (Sec. 2.1). **Finally**, many algorithmic tasks require combining a number of local computations in the right order. This typically implies that at a given time, attention should not focus on all possible matches, but only on the closest match. We propose and investigate a new type of attention with the corresponding inductive bias (Sec. 2.2), which we call *geometric attention*. We refer to the resulting new Transformer as Transformer Control Flow (TCF). In the experimental section (Sec. 3), we evaluate this model on three algorithmic tasks requiring length generalization and demonstrate its effectiveness.

2.1 Copy gate: learning to skip operations (vertical flow)

Each layer of the regular Transformer consists of one self-attention and one feedforward block. The input to each of these blocks are directly connected to the corresponding output via a residual connection (Srivastava et al., 2015; He et al., 2016). However, such a connection does not allow for skipping the transformation of the entire layer and simply passing the unchanged input to the next layer. Here we propose to add an explicit gate, which we call *copy gate*, to facilitate such a behavior.

We consider a T-layer Transformer encoder and an input sequence of length N. Since each layer corresponds to one *computational step*, we often refer to a layer as a step t. We denote the Transformer state of column i in layer t as $\boldsymbol{h}^{(i,t)} = \mathbf{H}_{t,i} \in \mathbb{R}^d$ where d is the state size, and $\mathbf{H}_t \in \mathbb{R}^{N \times d}$ denotes the states of all N columns in layer t. In the copy gate-augmented Transformer, each column i in layer $(t+1)$ processes the input $\mathbf{H}_t$ like in the regular Transformer:

$$\boldsymbol{a}^{(i,t+1)} = \mathrm{LayerNorm}(\mathrm{MultiHeadAttention}(\boldsymbol{h}^{(i,t)}, \mathbf{H}_t, \mathbf{H}_t) + \boldsymbol{h}^{(i,t)}) \tag{1}$$

$$\hat{\boldsymbol{h}}^{(i,t+1)} = \mathrm{LayerNorm}(\mathrm{FFN}^{\mathrm{data}}(\boldsymbol{a}^{(i,t+1)})) \tag{2}$$

but the output is gated as:

$$\boldsymbol{g}^{(i,t+1)} = \sigma(\mathrm{FFN}^{\mathrm{gate}}(\boldsymbol{a}^{(i,t+1)})) \tag{3}$$

$$\boldsymbol{h}^{(i,t+1)} = \boldsymbol{g}^{(i,t+1)} \odot \hat{\boldsymbol{h}}^{(i,t+1)} + (1 - \boldsymbol{g}^{(i,t+1)}) \odot \boldsymbol{h}^{(i,t)} \tag{4}$$

We use the basic two-layer feedforward block (Vaswani et al., 2017) for both $\mathrm{FFN}^{\mathrm{data}}$ and $\mathrm{FFN}^{\mathrm{gate}}$ which transforms input $\boldsymbol{x} \in \mathbb{R}^d$ to:

$$\mathrm{FFN}(\boldsymbol{x}) = \boldsymbol{W}_2 \max(\boldsymbol{W}_1 \boldsymbol{x} + \boldsymbol{b}_1, 0) + \boldsymbol{b}_2 \tag{5}$$

but with separate parameters and different dimensionalities: for $\mathrm{FFN}^{\mathrm{data}}$ $\boldsymbol{W}_1^{\mathrm{data}} \in \mathbb{R}^{d_{\mathrm{FF}} \times d}$, $\boldsymbol{W}_2^{\mathrm{data}} \in \mathbb{R}^{d \times d_{\mathrm{FF}}}$, while for $\mathrm{FFN}^{\mathrm{gate}}$ $\boldsymbol{W}_1^{\mathrm{gate}}, \boldsymbol{W}_2^{\mathrm{gate}} \in \mathbb{R}^{d \times d}$, with biases $\boldsymbol{b}_1^{\mathrm{data}} \in \mathbb{R}^{d_{\mathrm{FF}}}$ and $\boldsymbol{b}_2^{\mathrm{data}}, \boldsymbol{b}_1^{\mathrm{gate}}, \boldsymbol{b}_2^{\mathrm{gate}} \in \mathbb{R}^d$.

When the gate is closed i.e. $\boldsymbol{g}^{(i,t+1)} = 0$ in Eq. 4, the entire transformation is skipped and the input is copied over to the next layer $\boldsymbol{h}^{(i,t+1)} = \boldsymbol{h}^{(i,t)}$. Crucially, we parameterize the gate (Eq. 3) as a

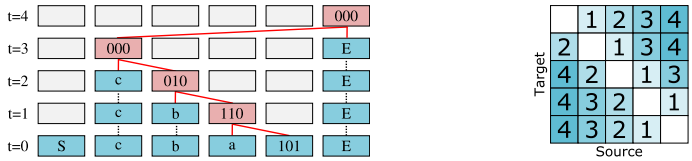

Figure 1: Left: an ideal sequence of computations in a Transformer for an example CTL task. Right: the order of source positions for each target, counting from the closest, used in geometric attention.

function of the output of the self-attention (Eq. 1), such that the decision to copy or to transform the input for each column depends on the states of all columns. This is a crucial difference compared to previously proposed gating in Transformers which are solely motivated by the training stability (Parisotto et al., 2020) or a common practice from convolution based models (Chaabouni et al., 2021).

The bias of the gate $\boldsymbol{b}_2^{\text{gate}}$ is initialized to -3 (Hochreiter & Schmidhuber, 1997). This ensures that initially no update happens, and creates a better gradient flow between layers. It also encourages the model to skip layers, unless they have an important contribution in the corresponding step.

2.2 Geometric attention: learning to attend to the closest match (horizontal flow)

We propose *geometric attention* designed to attend to the closest matching element. Like in regular self-attention, given an input sequence $[\boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}, ..., \boldsymbol{x}^{(N)}]$ with $\boldsymbol{x}^{(i)} \in \mathbb{R}^{d_{\text{in}}}$, each input is projected to key $\boldsymbol{k}^{(i)} \in \mathbb{R}^{d_{\text{key}}}$, value $\boldsymbol{v}^{(i)} \in \mathbb{R}^{d_{\text{value}}}$, query $\boldsymbol{q}^{(i)} \in \mathbb{R}^{d_{\text{key}}}$ vectors, and the dot product is computed for each key/query combination. In our geometric attention, the dot product is followed by a sigmoid function to obtain a score between 0 and 1:

$$\boldsymbol{P}_{i,j} = \sigma(\boldsymbol{k}^{(j)\top}\boldsymbol{q}^{(i)}) \tag{6}$$

which will be treated as a probability of the key at (source) position j matching the query at (target) position i. These probabilities are finally converted to the attention scores $\boldsymbol{A}_{i,j}$ as follows:

$$\boldsymbol{A}_{i,j} = \boldsymbol{P}_{i,j} \prod_{k \in \mathbb{S}_{i,j}} (1 - \boldsymbol{P}_{i,k}) \tag{7}$$

where $\mathbb{S}_{i,j}$ denotes the set of all (source) indices which are closer to i than j to i, and when two indices have the same distance to i, we consider the one which is greater than i to be closer, i.e.,

$$\mathbb{S}_{i,j} = \begin{cases} k \in \{1, ..., N\} \setminus \{i, j\} : |i-k| \leq |i-j|, & \text{if } i < j \\ k \in \{1, ..., N\} \setminus \{i, j\} : |i-k| < |i-j|, & \text{if } j < i \end{cases} \tag{8}$$

In addition, we explicitly zero-out the diagonal by setting $\boldsymbol{A}_{i,i} = 0$ for all $i = 1, ..., N$. The ordering of source indices is illustrated in Figure 1/Right. The resulting scores $\boldsymbol{A}_{i,j}$ are the attention scores used to compute the weighted average of the value vectors.

By using the term $(1 - \boldsymbol{P}_{i,k})$ in Eq. 7, when there is a match, it downscales any other matches which are more distant. Two recent works (Brooks et al., 2021; Banino et al., 2021) use such a parameterized geometric distribution in the form of Eq. 7 (see Sec. 6 on related work).

The resulting attention function has a complexity of $O(N^2)$, similar to the regular self-attention used in Transformers (Vaswani et al., 2017). Eq. 7 can be implemented in a numerically stable way in log-space. The products can then be calculated using cumulative sums, subtracting the elements for the correct indices in each position.

Directional encoding. In practice, we augment Eq. 6 by an additional *directional encoding*. In fact, the only positional information available in the geometric attention presented above is the ordering used to define the product in Eqs. 7-8. In practice, we found it crucial to augment the score computation of Eq. 6 with an additional *directional information*. The directional information we introduce is encoded as a scalar $\boldsymbol{D}_{i,j}$ for each target/source position pair (i, j) computed as:

$$\boldsymbol{D}_{i,j} = \begin{cases} \boldsymbol{W}_{\text{LR}}\boldsymbol{h}^{(i)} + b_{\text{LR}}, & \text{if } i \leq j \\ \boldsymbol{W}_{\text{RL}}\boldsymbol{h}^{(i)} + b_{\text{RL}}, & \text{if } i > j \end{cases} \tag{9}$$

where $\boldsymbol{h}^{(i)} \in \mathbb{R}^d$ denotes the input/state at position i and $\boldsymbol{W}_{\text{LR}}, \boldsymbol{W}_{\text{RL}} \in \mathbb{R}^{1\times d}$, $b_{\text{LR}}, b_{\text{RL}} \in \mathbb{R}$ are trainable parameters. This directional information is integrated into the score computation of Eq. 6 as follows (akin to how Dai et al. (2019) introduce the relative positional encoding as an extra term in the computation of attention scores):

$$\boldsymbol{P}_{i,j} = \sigma\big(\alpha\big(\boldsymbol{h}^{(i)}\boldsymbol{W}_q + \boldsymbol{u}\big)^\top \boldsymbol{W}_{k,E}\boldsymbol{h}^{(j)} + \beta \boldsymbol{D}_{i,j} + \gamma\big) \tag{10}$$

where the matrix $\boldsymbol{W}_q \in \mathbb{R}^{d_{\text{head}}\times d}$ maps the states to queries, $\boldsymbol{u} \in \mathbb{R}^{d_{\text{head}}}$ is a bias for queries, $\boldsymbol{W}_{k,E} \in \mathbb{R}^{d_{\text{head}}\times d}$ maps states to keys (we note that d_{head} is typically the size of the key, query and value vectors for each head, $d_{\text{head}} = \frac{d}{n_{\text{heads}}}$), and $\alpha, \beta, \gamma \in \mathbb{R}$ are learned scaling coefficients and bias, initialized to $\alpha = \frac{1}{\sqrt{d_{\text{head}}}}, \beta = 1, \gamma = 0$. Using this additional directional information, each query (position i) can potentially learn to restrict its attention to either the left or right side.

3 Experiments

We evaluate the proposed methods on three tasks: the standard compositional table lookup (Liska et al., 2018; Hupkes et al., 2019) and two tasks we propose: simple arithmetic and a variant of ListOps (Nangia & Bowman, 2018). In all cases, the task is designed to test NNs' ability for compositional generalization: the model has to learn to apply operations seen during training in a longer/deeper compositional form. Further experimental details for each task can be found in Appendix C.

3.1 Compositional table lookup

Task. Compositional table lookup task (Liska et al., 2018; Hupkes et al., 2019; Dubois et al., 2020) is constructed from a set of symbols and unitary functions defined over these symbols. Each example in the task is defined with one input symbol and a list of functions to be applied sequentially, i.e. the first function is applied to the input symbol, and the resulting output becomes the input to the second function, and so forth. There are 8 possible symbols. Each symbol is traditionally represented by a 3-bit bitstring (Liska et al., 2018), however, in practice, they are simply processed as one token (Dubois et al., 2020). The functions are bijective and randomly generated. Each function is represented by a letter. An example input is '`101 d a b`', which corresponds to the expression $b(a(d(101)))$; the model has to predict the correct output symbol. We note that there exists a sequence-to-sequence variant of this task (Dubois et al., 2020) where the model has to predict all intermediate steps (thus trained with intermediate supervision). We directly predict the final output. An ideal model should be able to solve this task independently of the presentation order, i.e., whether the task is encoded as '`101 d a b`' or '`b a d 101`'. We thus study both forward (former) and backward (latter) variants of the task. To evaluate systematic generalization, the train/valid/test sets have different numbers of compositions: samples of up to 5/6-8/9-10 operations respectively. No previous work has reported the perfect accuracy on this task using a NN. We refer the readers to Sec. 6 for further details on the previous work.

Results. We consider five different baselines: an LSTM (Hochreiter & Schmidhuber, 1997), bidirectional LSTM, DNC (Graves et al., 2016; Csordás & Schmidhuber, 2019), Universal Transformers (Vaswani et al., 2017; Dehghani et al., 2019), and its relative position variants (the variant from Csordás et al. (2021) to be specific). For Transformers, the prediction is based on the last column in the final layer (we conduct an ablation study on this choice in Appendix A). The hyper-parameters used for each model can be found in Table 7 in the appendix. We also provide an ablation study on the number of layers needed for generalization in Appendix A, which supports our claim on the necessity for a "sufficiently" deep architecture. The main results on this task are shown in Table 1. The LSTM and DNC perform well in the forward variant, achieving perfect generalization for longer sequences, but fail on the backward variant. This is not surprising since in the forward case, input symbols are presented in the right processing order to the LSTM. As expected, the bidirectional LSTM performs well in both presentation orders, since one of its processing directions is always aligned with the order of computation. However, for an arbitrary task, the order of processing is not given. For example, for ListOps (Sec. 3.3), the processing should start from the deepest point in the parse tree, which is probably somewhere in the middle of the sequence. The experiments on other tasks (Sec. 3.2 and 3.3) requiring arbitrary processing orders show that bidirectional LSTMs do not generalize well in such tasks. This is not satisfactory since our goal is to create a generic architecture which can solve arbitrary problems with an arbitrary underlying input processing order. While the

Transformer seems to be a good candidate for learning problem dependent processing orders, the baseline Transformer variants fail to generalize in this task in both directions.

By introducing the copy gate (Sec. 2.1), the relative Transformer is able to solve the forward task, but not the backward one. Our analysis showed that the network learns to attend to the last operation based on the relative position information. Since the result is read from the last column, this position changes with the sequence length. The model thus fails to generalize to such arbitrary offsets. To remediate this issue, we introduce a simple mechanism to let the model choose between absolute and relative positional encodings at each position (see Appendix B). The resulting model effectively manages to make use of the absolute position for the prediction, and to perform well in both directions. However, such a combination of absolute/relative positional encoding might be an overly specific bias. A more generic solution, geometric attention (Sec. 2.2), also achieves perfect generalization, and was found easier to train. We present the corresponding visualization of our model in Sec. 4.

Table 1: Accuracy on **compositional table lookup** dataset.

	IID		Longer	
Model	Forward	Backward	Forward	Backward
LSTM	**1.00 ± 0.00**	0.59 ± 0.03	**1.00 ± 0.00**	0.22 ± 0.03
Bidirectional LSTM	**1.00 ± 0.00**	**1.00 ± 0.00**	**1.00 ± 0.00**	**1.00 ± 0.00**
DNC	**1.00 ± 0.00**	0.57 ± 0.06	**1.00 ± 0.00**	0.18 ± 0.02
Transformer	**1.00 ± 0.00**	0.82 ± 0.39	0.13 ± 0.01	0.12 ± 0.01
+ rel	**1.00 ± 0.00**	**1.00 ± 0.00**	0.23 ± 0.05	0.13 ± 0.01
+ rel + gate	**1.00 ± 0.00**	**1.00 ± 0.00**	**0.99 ± 0.01**	0.19 ± 0.04
+ abs/rel + gate	**1.00 ± 0.00**	**1.00 ± 0.00**	**0.98 ± 0.02**	**0.98 ± 0.03**
+ geom. att.	0.96 ± 0.04	0.93 ± 0.06	0.16 ± 0.02	0.15 ± 0.02
+ geom. att. + gate (TCF)	**1.00 ± 0.00**	**1.00 ± 0.00**	**1.00 ± 0.00**	**1.00 ± 0.00**

3.2 Simple arithmetics

In order to validate the success of the proposed model on a task which involves more complex data flows and operations, we propose a *simple arithmetic* task.

Task. The task is to execute an arithmetic expression consisting of nested modulo 10 additions and multiplications. This requires the model to process tree-structured data flows, which is presumably more difficult than the sequential processing required for the CTL task. Each operation is surrounded by brackets, such that the boundaries of operations are easy to determine. For example '`((4*7)+2)`' should evaluate to '`0`' (30 modulo 10). The expressions are generated randomly. The tree depth is up to 5 for the training set, 6 for the validation and 7 and 8 for the test set. The depth is measured as the number of operations without the leaves, so the example above has a depth of 2. The length of all examples are limited to a maximum of 50 tokens.

Results. Table 2 shows the results. All considered models performs well on the IID validation data, but none except the TCF performs well on the generalization test set, which achieves a near-perfect accuracy of 98%. We also note that the TCF learns very fast: while all other models required about 200 K steps to converge, the TCF already achieves the near-perfect accuracy at 50 K steps of training.

3.3 ListOps

We also evaluate our model on a variant of the ListOps task (Nangia & Bowman, 2018) which is a popular task commonly used to evaluate parsing abilities of neural networks (Havrylov et al., 2019; Shen et al., 2019; Tay et al., 2021; Xiong et al., 2021; Irie et al., 2021). Some special architectures such as Chowdhury & Caragea (2021) can almost perfectly generalize to longer sequences on this task. However, as far as we know, no Transformer variant has been reported to be fully successful.

Task. The task consists of executing nested list operations written in prefix notation. The elements of a list are either one digit number (from 0 to 9) or recursively another list. The operations are

Table 2: Performance of different models on **simple arithmetic** dataset. All models are trained 200 K iterations, except the TCF which we stopped training at 100 K. We also report the performance of all models after 50 K iterations, where it can be seen that our model converges significantly faster compared to the others.

	IID (1..5)	Test (7..8)	
	200 K	200 K	50 K
LSTM	**0.99** ± **0.00**	0.74 ± 0.02	0.72 ± 0.01
Bidirectional LSTM	**0.98** ± **0.01**	0.82 ± 0.06	0.80 ± 0.04
Transformer	**0.98** ± **0.01**	0.47 ± 0.01	0.29 ± 0.01
+ rel	**1.00** ± **0.00**	0.77 ± 0.04	0.40 ± 0.05
+ abs/rel + gate	**1.00** ± **0.01**	0.80 ± 0.16	0.73 ± 0.15
+ geom. att. + gate (TCF)	**1.00** ± **0.00**	**0.98** ± **0.01**	**0.98** ± **0.01**

min, max, median and sum. The sum is modulo 10 and the median is followed by the floor function such that the output of any function/operation is between 0 and 9. For example: `[MED 4 8 5 [MAX 8 4 9 ] ]` should return `6`. There are two well known variants of ListOps: the original version by Nangia & Bowman (2018) and the "Long Range Arena" version by Tay et al. (2021) which differ from each other in terms of the maximum number of arguments in each function and the maximum sequence length in terms of number of tokens. In both versions, there is no strict control on the depth of samples in the data: there is simply a certain pre-defined probability such that each position/element in the list is expanded to start another list (which can increase the tree depth). This is not suitable for evaluating systematic generalization in terms of compositionality (over the problem depth). We instead propose to generate clean train, valid, and test splits with disjoint depths. In our settings, the training set contains samples with depths of up to 5, the validation set only contains the depth-7 samples and finally, the test set contains depth-7 and 8 samples. Also importantly, we make sure that a depth-K sample effectively requires computation until the depth-K (without such a control, min, max, and med operations can potentially find the output without checking/executing all its arguments). By dissociating the splits by the depth, we can clearly identify models which fail to compositionally generalize. Apart from the depth specifications, all train/valid/test sets share the same following settings: the maximum sequence length is 50 tokens, the probability of recursively sampling another function inside a list is 30% at each position, and the maximum number of arguments for a function is 5. The train set consists of 1M, the valid and test sets of 1K sequences.

Results. Table 3 shows the results. Similarly to compositional table lookup and simple arithmetic tasks, the baseline LSTM and Transformers do not generalize well to the test set consisting of deeper problems, while they achieve a near perfect accuracy on IID data. In contrast, our model achieves near-perfect generalization.

Table 3: Performance of different models on balanced **ListOps** dataset. All models are trained 200 K iterations, except all `+gate` variants which converge after 100 K steps.

	IID (1..5)	Test (7..8)
LSTM	**0.99** ± **0.00**	0.71 ± 0.03
Bidirectional LSTM	**1.00** ± **0.00**	0.57 ± 0.04
Transformer	0.98 ± 0.00	0.74 ± 0.03
+ rel	**0.98** ± **0.01**	0.79 ± 0.04
+ abs/rel + gate	**1.00** ± **0.01**	0.90 ± 0.06
+ geom. att. + gate (TCF)	**1.00** ± **0.00**	**0.99** ± **0.01**

4 Analysis

In this section, we provide some visualizations of attention and gating patterns of the TCF and the corresponding analyses. For more visualizations, we refer the readers to Appendix D.

Compositional Table Lookup. Figure 2 shows the gating and attention patterns of the TCF model for an example of the backward presentation task. As shown in Fig. 2/Bottom, the gates of different columns open sequentially one after another, when the input is available for them. Fig. 2/Top shows the corresponding attention maps. Each column attends to the neighboring one, waiting for its computation to be finished. The behavior of the last column is different: it always attends to the second position of the sequence, which corresponds to the last operation to be performed.

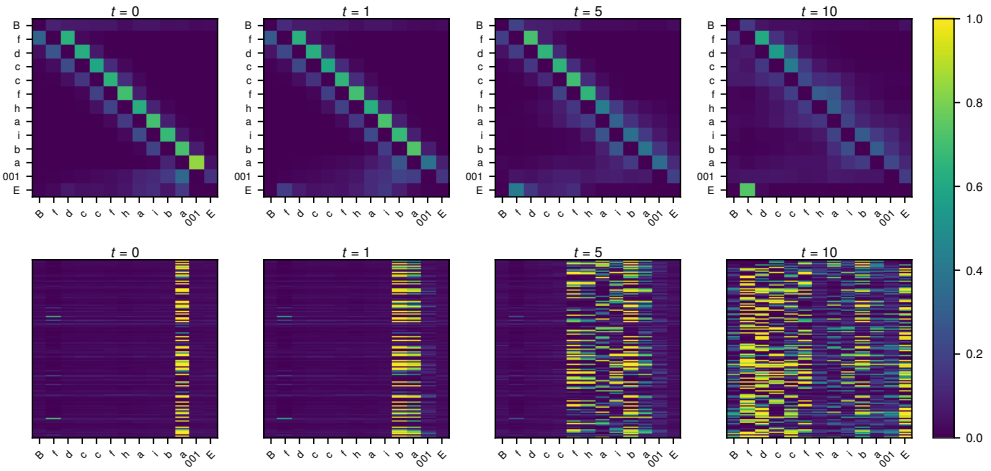

Figure 2: Example visualization of TCF. For other models, see Appendix D. Top: Attention map for different steps. The x/y-axis corresponds to source/target positions respectively. Each position focuses on the column to the right, except the last one where the result is read from, which focuses on the last operation. The focus becomes clear only once the result is available. Bottom: gate activations for different steps/layers. The gates remain closed until the data dependencies are satisfied.

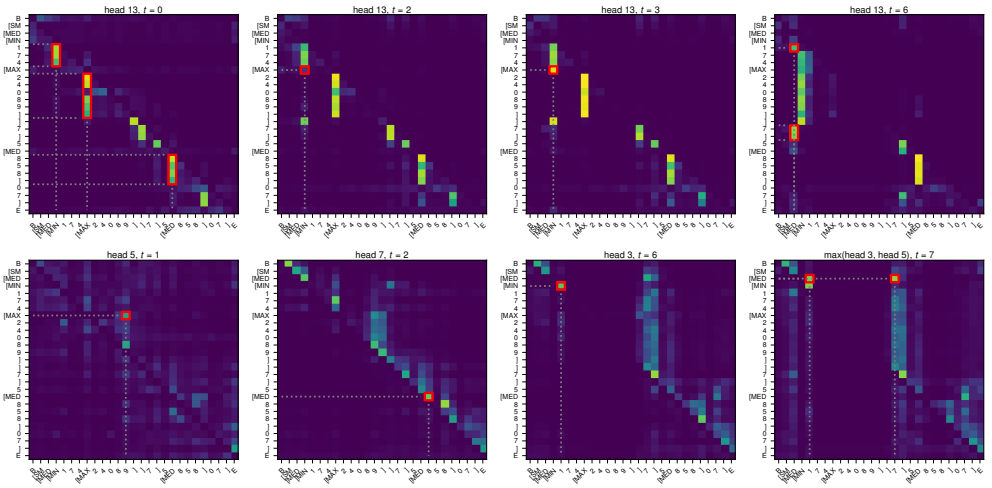

Figure 3: Example visualization of TCF on ListOps. The top row shows *head 13* in different steps, which controls which arguments used in which step. The bottom row shows different heads in different key steps. Please refer to Sec. 4 for the step-by-step description. More visualizations are provided in the appendix: Fig. 10 shows the max of attention over all heads for all steps, Fig. 11 shows all steps of *head 13*, and Fig. 12 shows the corresponding gates.

ListOps. We can also identify how the data is processed by the TCF in ListOps. We observe that different attention heads play different roles. We highlight the core observations in Figure 3. The input for this example is: `[SM [MED [MIN 1 7 4 [MAX 2 4 0 8 9 ] ] 7 ] 5 [MED 8 5 8 ] 0 7 ]`. First of all, we find that there is a head (*head 13* in Figure 3, *first row*) which seems to

be responsible for connecting operators and their arguments: the operands/arguments of an operation attend to the operator. In step 0 ($t = 0$ in the figure), we can recognize that the operations at the deepest level, namely `MAX` and the second `MED` have all the arguments ready (as is shown by vertical lines on the columns corresponding to `MAX` and `MED`). The model certainly identifies that these two operations are ready to be processed and that they can be processed in parallel (these arguments-to-operation attention patterns remain for a few steps). We note that at this stage, the last arguments of `MIN` is not ready yet (`[MIN 1 7 4 [MAX 2 4 0 8 9 ] ]`). We can see that only arguments which are already ready (`1 7 4`) attend to the operator (see the column of `MIN`). In step 1 ($t = 1$, *2nd row*), we can see that *head 5* copies the expected result of `MAX`, `9` to the column of the operator (we note that certainly this only requires one step as `9` is always the result of `MAX` when it is one of the arguments of `MAX`). Similarly in step 2, *head 7* (*2nd row*) seems to copy the result of the second `MED`, `8` to the operator column. In step 3 ($t = 3$, *1st row*), we can recognize that the result of `MAX` is marked as an argument for `MIN` in *head 13* which is responsible for communication between operators and its arguments. This is shown by the new attention which appears at $t = 3$ in *head 13* from the source position `MAX` to the target position `MIN` (a pattern which is not visible at $t = 2$). In *head 3*, $t = 6$ (*2nd row*), the expected result of `MIN`, which is `1`, is copied to the operator, similarly to the patterns we observed above for `MAX` and `MED`. In *head 13*, $t = 6$ (*1st row*), we can observe that all arguments for the first `MED` are also now recognized (the result of `MIN` which is `1`, and `7`). Finally in $t = 7$ (*2nd row*), two heads, *head 3* and *head 5* seem to copy/gather two inputs needed to compute the corresponding median, `1` and `7`, to the operator column. A complete visualization for further steps can be found in Appendix D.2. we noticed that some of the heads do not seem to play a key role. Here we focused on interpreting those which seem to participate in the main computation. For ListOps, we note that we also partially find the attention patterns described above in the baseline Transformer with relative positional encoding, at least on some examples we have inspected, which also explains its rather high accuracy.

5 Discussion

Learning adaptive serialization. The TCF architecture could be also understood as doing adaptive serialization of the problem. A key requirement for reusable computation is the decomposition of the problem in reusable building blocks, typically in sequential steps. The granularity of the decomposition determines the reusability: fusing operations in a single step makes the processing faster (less steps), but also more specialized. Learning the most granular solutions is thus preferable for generalization. At the same time, not all processing should happen serially: branches of the computation graph that does not have common data dependencies can be processed independently in parallel, which we empirically observe in our TCF in the ListOps example (Sec. 4). This enables the architecture to get away with the number of computation steps reflecting the depth of the computation graph instead of the length of the input.

Bottom up approach for improving model architectures. Transformers have seen tremendous successes across various application domains (Devlin et al., 2019; Brown et al., 2020; Dosovitskiy et al., 2021). Impressive results have been reported when they are scaled up with a large amount of data (Brown et al., 2020). At the same time, there are simple tasks like those we highlight in this work, which demonstrate that the Transformer architecture still has to be improved. In fact, in algorithmic tasks, it is often the case that a sub-optimal choice of the architecture/optimization method makes the model fall back to memorization. We argue that it is crucial to look at isolated problems which test specific generalization capability. This calls for approaching the problem bottom up: building on toy tasks that focus on individual aspects of generalization, and using them for improving models.

6 Related work

Gating inside Transformers. Several prior works have proposed to use some form of gating inside Transformer architectures. However, the *copy gate* we propose is different from those previously proposed (Parisotto et al., 2020; Chaabouni et al., 2021) as it satisfies two important properties. First, our copy gate allows the model to skip the *entire* Transformer layer (i.e both the self-attention and the feedforward blocks) when the gate is closed. Second, the gate function is conditioned on the attention output such that the decision of opening or closing depends on the information from all columns. While Parisotto et al. (2020) have proposed multiple gating variants in the goal of stabilizing

Transformers for reinforcement learning, none of the proposed approaches can produce this behavior. Empirically, we also tried out a few other gating variants which do not satisfy the two properties above; we found them not to improve over the regular Transformers in our preliminary experiments on compositional table lookup. Chaabouni et al. (2021) also make use of "gating" in Transformers. To be more specific, they use the gated linear unit activation function (Dauphin et al., 2017) commonly used in convolutional NNs. They import such an activation function to their Transformer models as it is part of the model (Dessì & Baroni, 2019) which has been reported to outperform RNN baselines on a systematic generalization task. Unlike our copy gate or Parisotto et al. (2020)'s gating, such a gating activation does not have the "residual" term (i.e. a closed gate zeros out the input) which allows the model to skip a transformation.

Parameterized geometric distributions. Two recent works (Brooks et al., 2021; Banino et al., 2021) have used a form of parameterized geometric distribution (PGD; in the form of Eq. 7). Brooks et al. (2021) have used such a geometric distribution to parameterize the movement of a pointer on a sequence of instructions. Banino et al. (2021) have used it to implement adaptive computation time (Schmidhuber, 2012; Graves, 2016). We use the PGD to obtain a generic attention mechanism as a replacement to the standard self-attention used in Transformers (Vaswani et al., 2017).

Compositional table lookup. CTL has been proposed by as a task for evaluating compositional ability of neural networks (Liska et al., 2018). Previous works evaluated RNNs, RNNs with attention, and Transformer architectures on this task with limited success (Hupkes et al., 2019; Dubois et al., 2020). Dubois et al. (2020) have proposed a special attention mechanism to augment the recurrent architecture. While they obtain good performance on the forward presentation order, the proposed model has still failed in the backward presentation order. In contrast, two of the approaches we proposed (Sec. 3.1) achieve 100% generalization accuracy on this task in both presentation orders.

Positional encodings. Many previous works have focused on improving positional encoding for self-attention. Most notably, the relative positional encoding (Shaw et al., 2018; Dai et al., 2019) has been found useful in improving systematic generalization of Transformers (Csordás et al., 2021). In this work, we also present two new approaches related to positional encoding. One is the gated combination of absolute and relative positional encoding (Sec. 3.1; details in Appendix B). We show that absolute positional encoding can complement the relative positional encoding. It enables the model to always attend to a specific position, as is needed for CTL task in the last step, while the gating allows it to use relative positional encoding for other positions/steps. Second, we introduce directional encoding to augment the geometric attention. Unlike positional encoding which can overfit to a range of positions seen during training, the direction information is found to be robust and to be a crucial augmentation to the geometric attention.

7 Conclusion

We proposed a new view on the internal operations of Transformer encoders, introducing a dynamic dataflow architecture between Transformer columns. This overcomes two shortcomings of traditional Transformers: the problem of keeping and routing data in unaltered fashion, which we solve by an additional copy gate, and the problem of learning length-independent attention patterns, which we solve by geometric attention. Our new model, TCF, generalizes to longer lengths on the popular compositional lookup table task in both forward and backward directions. TCF also achieves near perfect performance in simple arithmetic and ListOps tasks in settings which test systematic generalization in terms of computational depth. In general, the gates and the attention maps collectively make the architecture more interpretable compared to the baselines. Future work will extend this encoder-only architecture to a full sequence-to-sequence model and evaluate it on other standard tasks in systematic generalization requiring generation of variable-length output sequences.

Reproducibility Statement. We will release all code and datasets used to produce the results reported in this work in a public repository. The corresponding code is already included in the supplemental material of this submission, including the code used to produce tables and plots presented in this work.

REFERENCES

Dzmitry Bahdanau, Harm de Vries, Timothy J O'Donnell, Shikhar Murty, Philippe Beaudoin, Yoshua Bengio, and Aaron Courville. CLOSURE: Assessing systematic generalization of CLEVR models. In *ViGIL workshop, NeurIPS*, Vancouver, Canada, December 2019.

Andrea Banino, Jan Balaguer, and Charles Blundell. Pondernet: Learning to ponder. *Preprint arXiv:2107.05407*, 2021.

Ethan A. Brooks, Janarthanan Rajendran, Richard L. Lewis, and Satinder Singh. Reinforcement learning of implicit and explicit control flow instructions. In *Proc. Int. Conf. on Machine Learning (ICML)*, pp. 1082–1091, Virtual only, July 2021.

Tom B Brown et al. Language models are few-shot learners. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, Virtual only, December 2020.

Rahma Chaabouni, Roberto Dessì, and Eugene Kharitonov. Can transformers jump around right in natural language? assessing performance transfer from scan. *Preprint arXiv:2107.01366*, 2021.

Xinyun Chen, Chen Liang, Adams Wei Yu, Dawn Song, and Denny Zhou. Compositional generalization via neural-symbolic stack machines. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, Virtual only, December 2020.

Jishnu Ray Chowdhury and Cornelia Caragea. Modeling hierarchical structures with continuous recursive neural networks. In *Proc. Int. Conf. on Machine Learning (ICML)*, pp. 1975–1988, Virtual only, July 2021.

Róbert Csordás and Jürgen Schmidhuber. Improving differentiable neural computers through memory masking, de-allocation, and link distribution sharpness control. In *Int. Conf. on Learning Representations (ICLR)*, New Orleans, LA, USA, May 2019.

Róbert Csordás, Kazuki Irie, and Jürgen Schmidhuber. The devil is in the detail: Simple tricks improve systematic generalization of transformers. In *Proc. Conf. on Empirical Methods in Natural Language Processing (EMNLP)*, Punta Cana, Dominican Republic, November 2021.

Zihang Dai, Zhilin Yang, Yiming Yang, Jaime G Carbonell, Quoc Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. In *Proc. Association for Computational Linguistics (ACL)*, pp. 2978–2988, Florence, Italy, 2019.

Yann N Dauphin, Angela Fan, Michael Auli, and David Grangier. Language modeling with gated convolutional networks. In *Proc. Int. Conf. on Machine Learning (ICML)*, pp. 933–941, Sydney, Australia, August 2017.

Mostafa Dehghani, Stephan Gouws, Oriol Vinyals, Jakob Uszkoreit, and Lukasz Kaiser. Universal transformers. In *Int. Conf. on Learning Representations (ICLR)*, New Orleans, LA, USA, May 2019.

Roberto Dessì and Marco Baroni. CNNs found to jump around more skillfully than RNNs: Compositional generalization in seq2seq convolutional networks. In *Proc. Association for Computational Linguistics (ACL)*, pp. 3919–3923, Florence, Italy, July 2019.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In *Proc. North American Chapter of the Association for Computational Linguistics on Human Language Technologies (NAACL-HLT)*, pp. 4171–4186, Minneapolis, MN, USA, June 2019.

Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In *Int. Conf. on Learning Representations (ICLR)*, Virtual only, May 2021.

Yann Dubois, Gautier Dagan, Dieuwke Hupkes, and Elia Bruni. Location attention for extrapolation to longer sequences. In *Proc. Association for Computational Linguistics (ACL)*, pp. 403–413, Virtual only, July 2020.

Jerry Fodor and Brian P McLaughlin. Connectionism and the problem of systematicity: Why smolensky's solution doesn't work. *Cognition*, 35(2):183–204, 1990.

Jerry A Fodor, Zenon W Pylyshyn, et al. Connectionism and cognitive architecture: A critical analysis. *Cognition*, 28(1-2):3–71, 1988.

Daniel Furrer, Marc van Zee, Nathan Scales, and Nathanael Schärli. Compositional generalization in semantic parsing: Pre-training vs. specialized architectures. *Preprint arXiv:2007.08970*, 2020.

Alex Graves. Adaptive computation time for recurrent neural networks. In *Int. Conf. on Learning Representations (ICLR) Workshop Track*, Vancouver, Canada, April 2016.

Alex Graves, Greg Wayne, Malcolm Reynolds, Tim Harley, Ivo Danihelka, Agnieszka Grabska-Barwinska, Sergio Gomez Colmenarejo, Edward Grefenstette, Tiago Ramalho, John P. Agapiou, Adrià Puigdomènech Badia, Karl Moritz Hermann, Yori Zwols, Georg Ostrovski, Adam Cain, Helen King, Christopher Summerfield, Phil Blunsom, Koray Kavukcuoglu, and Demis Hassabis. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626): 471–476, 2016.

Klaus Greff, Sjoerd van Steenkiste, and Jürgen Schmidhuber. On the binding problem in artificial neural networks. *Preprint arXiv:2012.05208*, 2020.

Stephen José Hanson. A stochastic version of the delta rule. *Physica D: Nonlinear Phenomena*, 42 (1-3):265–272, 1990.

Serhii Havrylov, Germán Kruszewski, and Armand Joulin. Cooperative learning of disjoint syntax and semantics. In *Proc. North American Chapter of the Association for Computational Linguistics on Human Language Technologies (NAACL-HLT)*, pp. 1118–1128, Minneapolis, USA, June 2019.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proc. The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, Las Vegas, NV, USA, June 2016.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, pp. 1735–1780, 1997.

Dieuwke Hupkes, Anand Singh, Kris Korrel, German Kruszewski, and Elia Bruni. Learning compositionally through attentive guidance. In *Proc. Int. Conf. on Computational Linguistics and Intelligent Text Processing*, La Rochelle, France, April 2019.

Dieuwke Hupkes, Verna Dankers, Mathijs Mul, and Elia Bruni. Compositionality decomposed: How do neural networks generalise? *Journal of Artificial Intelligence Research*, pp. 757–795, 2020.

Kazuki Irie, Imanol Schlag, Róbert Csordás, and Jürgen Schmidhuber. Going beyond linear transformers with recurrent fast weight programmers. *Preprint arXiv:2106.06295*, 2021.

Daniel Keysers, Nathanael Schärli, Nathan Scales, Hylke Buisman, Daniel Furrer, Sergii Kashubin, Nikola Momchev, Danila Sinopalnikov, Lukasz Stafiniak, Tibor Tihon, Dmitry Tsarkov, Xiao Wang, Marc van Zee, and Olivier Bousquet. Measuring compositional generalization: A comprehensive method on realistic data. In *Int. Conf. on Learning Representations (ICLR)*, Addis Ababa, Ethiopia, 2020.

Kris Korrel, Dieuwke Hupkes, Verna Dankers, and Elia Bruni. Transcoding compositionally: Using attention to find more generalizable solutions. In *Proc. BlackboxNLP Workshop on Analyzing and Interpreting Neural Networks for NLP, ACL*, pp. 1–11, Florence, Italy, 2019.

Brenden M Lake. Compositional generalization through meta sequence-to-sequence learning. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, pp. 9788–9798, Vancouver, Canada, December 2019.

Brenden M. Lake and Marco Baroni. Generalization without systematicity: On the compositional skills of sequence-to-sequence recurrent networks. In *Proc. Int. Conf. on Machine Learning (ICML)*, pp. 2873–2882, Stockholm, Sweden, July 2018.

Yuanpeng Li, Liang Zhao, Jianyu Wang, and Joel Hestness. Compositional generalization for primitive substitutions. In *Proc. Conf. on Empirical Methods in Natural Language Processing and Int.Joint Conf. on Natural Language Processing (EMNLP-IJCNLP)*, pp. 4292–4301, Hong Kong, China, November 2019.

Adam Liska, Germán Kruszewski, and Marco Baroni. Memorize or generalize? searching for a compositional RNN in a haystack. In *AEGAP Workshop ICML*, Stockholm, Sweden, July 2018.

Qian Liu, Shengnan An, Jian-Guang Lou, Bei Chen, Zeqi Lin, Yan Gao, Bin Zhou, Nanning Zheng, and Dongmei Zhang. Compositional generalization by learning analytical expressions. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, Virtual only, December 2020.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *Int. Conf. on Learning Representations (ICLR)*, New Orleans, LA, USA, May 2019.

Nikita Nangia and Samuel R. Bowman. Listops: A diagnostic dataset for latent tree learning. In *Proc. North American Chapter of the Association for Computational Linguistics on Human Language Technologies (NAACL-HLT)*, pp. 92–99, New Orleans, USA, June 2018.

Santiago Ontañón, Joshua Ainslie, Vaclav Cvicek, and Zachary Fisher. Making transformers solve compositional tasks. *Preprint arXiv:2108.04378*, 2021.

Emilio Parisotto, H. Francis Song, Jack W. Rae, Razvan Pascanu, Çaglar Gülçehre, Siddhant M. Jayakumar, Max Jaderberg, Raphaël Lopez Kaufman, Aidan Clark, Seb Noury, Matthew Botvinick, Nicolas Heess, and Raia Hadsell. Stabilizing transformers for reinforcement learning. In *Proc. Int. Conf. on Machine Learning (ICML)*, volume 119, pp. 7487–7498, Virtual only, July 2020.

Jake Russin, Jason Jo, Randall C O'Reilly, and Yoshua Bengio. Compositional generalization in a deep seq2seq model by separating syntax and semantics. *Preprint arXiv:1904.09708*, 2019.

Jürgen Schmidhuber. Self-delimiting neural networks. Technical Report IDSIA-08-12, arXiv:1210.0118v1, The Swiss AI Lab IDSIA, 2012.

Peter Shaw, Jakob Uszkoreit, and Ashish Vaswani. Self-attention with relative position representations. In *Proc. North American Chapter of the Association for Computational Linguistics on Human Language Technologies (NAACL-HLT)*, pp. 464–468, New Orleans, Louisiana, USA, June 2018.

Peter Shaw, Ming-Wei Chang, Panupong Pasupat, and Kristina Toutanova. Compositional generalization and natural language variation: Can a semantic parsing approach handle both? *Preprint arXiv:2010.12725*, 2020.

Yikang Shen, Shawn Tan, Seyed Arian Hosseini, Zhouhan Lin, Alessandro Sordoni, and Aaron C. Courville. Ordered memory. In *Proc. Advances in Neural Information Processing Systems (NeurIPS)*, pp. 5038–5049, Vancouver, Canada, December 2019.

Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

Rupesh K Srivastava, Klaus Greff, and Jürgen Schmidhuber. Training very deep networks. In *Proc. Advances in Neural Information Processing Systems (NIPS)*, pp. 2368–2376, Montreal, Canada, December 2015.

Yi Tay, Mostafa Dehghani, Samira Abnar, Yikang Shen, Dara Bahri, Philip Pham, Jinfeng Rao, Liu Yang, Sebastian Ruder, and Donald Metzler. Long range arena : A benchmark for efficient transformers. In *Int. Conf. on Learning Representations (ICLR)*, Virtual only, May 2021.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Proc. Advances in Neural Information Processing Systems (NIPS)*, pp. 5998–6008, Long Beach, CA, USA, December 2017.

Gail Weiss, Yoav Goldberg, and Eran Yahav. Thinking like Transformers. In *Proc. Int. Conf. on Machine Learning (ICML)*, pp. 11080–11090, Virtual only, July 2021.

Yunyang Xiong, Zhanpeng Zeng, Rudrasis Chakraborty, Mingxing Tan, Glenn Fung, Yin Li, and Vikas Singh. Nyströmformer: A Nyström-based algorithm for approximating self-attention. In *Proc. AAAI Conf. on Artificial Intelligence*, pp. 14138–14148, Virtual only, February 2021.

A ABLATIONS

Number of layers. In Sec. 2, we hypothesized that decomposition of the problem into its elementary operations is a necessary property of a model which generalizes. This motivated us to configure our models to have at least as many layers as the depth of the computation involved, plus a few additional layers for writing the output and for gathering an overview of the problem at the beginning. We assumed that in such a model with a sufficient number of layers, each layer learns the underlying "elementary" operation. The resulting models are thus deeper than those typically used in the literature for similar tasks (Keysers et al., 2020; Tay et al., 2021). Here we provide an ablation study to demonstrate that such depths are effectively necessary for generalization. We measure the IID and generalization performance with various numbers of layers on the compositional table lookup dataset. Since our test set on the CTL task consists of up to 10 function applications, it should require about 12 layers according to our hypothesis. Table 4 shows the results. We clearly observe that, while the shallow models can also solve the IID split, only the deep models generalize to the longer problems (here the 12-layer model generalizes almost perfectly, but the 10-layer one doesn't).

Table 4: The performance of TCF on the **compositional table lookup** dataset, with different number of layers.

	IID		Test	
n_{layers}	Forward	Backward	Forward	Backward
14	1.00 ± 0.00	1.00 ± 0.00	**1.00 ± 0.00**	**1.00 ± 0.00**
12	1.00 ± 0.00	1.00 ± 0.00	**1.00 ± 0.00**	**0.99 ± 0.02**
10	1.00 ± 0.00	1.00 ± 0.00	0.75 ± 0.04	0.62 ± 0.05
8	1.00 ± 0.00	1.00 ± 0.00	0.23 ± 0.02	0.24 ± 0.03
6	1.00 ± 0.00	0.96 ± 0.03	0.22 ± 0.05	0.15 ± 0.01
4	0.96 ± 0.04	0.68 ± 0.11	0.14 ± 0.01	0.13 ± 0.01

Readout from the first instead of the last column. In our experiments with the Transformer models, the last column was used for the readout of the result. Under this configuration, the readout position depends on the length of the sequence which might increase the difficulty of the problem, in particular for the models using absolute positional embeddings. Table 5 shows the corresponding ablation study. We observe that this choice has only marginal impact on the model performance. As a side note, we also tried the variant where an additional cross-attention layer is used for the readout. Again, the generalization performance was not better. In fact, these results are not surprising since none of these changes fundamentally addresses the problem of length generalization.

Table 5: Accuracy on **compositional table lookup** dataset with the results read from the first or last column (**Readout**).

		IID		Longer	
Model	Readout	Forward	Backward	Forward	Backward
Transformer	First	1.00 ± 0.00	0.82 ± 0.39	0.12 ± 0.01	0.13 ± 0.01
	Last	1.00 ± 0.00	0.82 ± 0.39	0.13 ± 0.01	0.12 ± 0.01
+ rel	First	1.00 ± 0.00	1.00 ± 0.00	0.12 ± 0.01	0.22 ± 0.05
	Last	1.00 ± 0.00	1.00 ± 0.00	0.23 ± 0.05	0.13 ± 0.01
+ rel + gate	First	1.00 ± 0.00	1.00 ± 0.00	0.17 ± 0.02	1.00 ± 0.00
	Last	1.00 ± 0.00	1.00 ± 0.00	0.99 ± 0.01	0.19 ± 0.04

Does Adaptive Computation Time (ACT) help? In this work, we determined the number of layers/steps to be used in the model based on heuristics (see Appendix C.1). We could also consider using Adaptive Computation Time (ACT) to dynamically determine the number of steps. Furthermore, ACT introduces a form of gating which creates shortcuts in the credit assignment path between the output and a result of an intermediate layer. This "copying" mechanism resulting from the ACT (i.e. stop computation at a certain time and copy the result to the output) is fundamentally different

from our copy gate (Sec. 2.1). Our copy gate allows Transformer columns to keep the input unchanged until it's their turn to be processed (a crucial property to implement control flow like behavior). This behavior can not be simulated by the ACT. Here we provide some experimental results on models with ACT which confirm that the proposed copy gate is a crucial component for generalization which can not be replaced by ACT.

We note that there are various versions of ACT in the literature, e.g., the variant used by Dehghani et al. (2019) in Universal Transformers is different from the one used by Graves (2016). Here we focus on two variants: one in which we directly apply Graves (2016) to Transformers, and another one used by Dehghani et al. (2019). We start with the description of the former.

An extra sigmoidal unit $\hat{p}^{(i,t)}$ is computed for each column i in each timestep t as:

$$\hat{p}^{(i,t)} = \sigma(\boldsymbol{W}_{\mathrm{H}} \boldsymbol{h}^{(i,t)} + b_{\mathrm{H}}) \tag{11}$$

where $\boldsymbol{W}_{\mathrm{H}} \in \mathbb{R}^{1 \times d}$ and $\boldsymbol{b}_{\mathrm{H}} \in \mathbb{R}$ are trainable parameters. By comparing the cumulative sum of $\hat{p}^{(i,t)}$ over time steps to a threshold value $(1 - \epsilon)$ with a hyper-parameter ϵ (0.01 in our experiment), we determine the termination step T^i as:

$$T^i = \min\{T_{\text{max}}, \min\{t' : \sum_{t=1}^{t'} \hat{p}^{(i,t)} \geq 1 - \epsilon\}\} \tag{12}$$

where T_{max} is the pre-defined maximum number of steps.

The corresponding *halting probability* $\hat{p}^{(i,t)}$ is then computed as:

$$R^i = 1 - \sum_{t=1}^{T^i - 1} \hat{p}^{(i,t)} \tag{13}$$

$$p^{(i,t)} = \begin{cases} R^i & \text{if } t = T^i \\ \hat{p}^{(i,t)} & \text{if } t < T^i \end{cases} \tag{14}$$

which is used to compute the final output of column i as:

$$\boldsymbol{o}^i = \sum_{t=1}^{T^i} p^{(i,t)} \boldsymbol{h}^{(i,t)} \tag{15}$$

In Dehghani et al. (2019)'s variant, a different equation is used in lieu of Eq. 15 above and the computation of the reminder term R^i in Eq. 13 above is not properly handled in case where Eq. 12 terminates because of the first condition on T_{max}. For further details, we refer the readers to Listing 1 and 2 in Dehghani et al. (2019) and/or our public code.

One subtlety introduced by Dehghani et al. (2019) which we note here is that the computation of the final output $\boldsymbol{o}^i$ of column i effectively "halts" after T^i (since $\boldsymbol{o}^i$ only depends on $\boldsymbol{h}^{(i,t)}$ for $0 < t < T^i$), but column i itself still continues transforming the hidden states $\boldsymbol{h}^{(i,t)}$ for steps $t > T^i$ until all columns reach the termination step, and its updated states can be attended/read by another column j which has not halted yet (i.e. $T^j > T^i$). In this sense, computation is never stopped independently for each column. The mechanism described above instead finds the *readout* steps for each column (as is used in Eq. 15). We follow this decision in our implementation of both variants.

In addition, a new regularizer term, $L_{\text{ACT}} = \alpha \frac{1}{N} \sum_{i=1}^{N} R^i$ is added to the loss function, where N is the length of the input sequence. This makes the network prefer short computations. We ran a hyper-parameter search for α from the following values: 0.001, 0.003, 0.01, 0.03, 0.1. We found $\alpha = 0.03$ to work the best.

We conducted experiments on the compositional table lookup task. We first noted that ACT helps training our baseline Transformer models with a maximum step of 14 layers which was not possible without ACT (our baseline Transformer had only 11 layers for this reason; see Table 7). The shortcut in the credit assignment path introduced by ACT certainly helps training of this 14 layer model. As we noticed that the models with ACT learn slower than those with gating, we increased the number of training steps to 60k steps which is twice as many as 30k used for the models without ACT. Table 6

shows the results. We observe that, interestingly, ACT enables generalization for longer lengths in the forward direction of the Transformer with relative positional embedding and the one with geometric attention. However, we were not able to find any configuration that generalizes in the backward case. This demonstrates that the copy gate is effectively a crucial component for generalization which can not be replaced by ACT. Furthermore, the models with ACT converge significantly slower and they are more unstable and very sensitive to the value of α even in the successful forward case. Overall, the only benefit of ACT is thus the adaptive depth, as is illustrated in Figure 4, which is orthogonal to our study.

Table 6: Accuracy on **compositional table lookup** dataset with adaptive computation time (ACT). Two variants of ACT are shown: "U" corresponds to Dehghani et al. (2019), while "A" is the variant described in Appendix A. We also include baselines without ACT from Table 1 as a reference. Generalization performance after 30k and 60k **training steps** are shown.

		IID		Longer, 60k		Longer, 30k	
Model	ACT	Forward	Backward	Forward	Backward	Forward	Backward
Transformer		1.00 ± 0.00	0.82 ± 0.39	-	-	0.13 ± 0.01	0.12 ± 0.01
	A	1.00 ± 0.00	1.00 ± 0.00	0.12 ± 0.02	0.12 ± 0.01	0.13 ± 0.01	0.13 ± 0.01
	U	1.00 ± 0.00	1.00 ± 0.00	0.12 ± 0.01	0.11 ± 0.01	0.13 ± 0.01	0.12 ± 0.01
+ rel		1.00 ± 0.00	1.00 ± 0.00	-	-	0.23 ± 0.05	0.13 ± 0.01
	A	1.00 ± 0.00	1.00 ± 0.00	0.99 ± 0.02	0.13 ± 0.00	0.84 ± 0.22	0.13 ± 0.01
	U	1.00 ± 0.00	1.00 ± 0.00	0.92 ± 0.14	0.12 ± 0.02	0.67 ± 0.41	0.12 ± 0.00
+ geo		0.96 ± 0.04	0.93 ± 0.06	-	-	0.16 ± 0.02	0.15 ± 0.02
	A	1.00 ± 0.00	1.00 ± 0.00	0.97 ± 0.05	0.45 ± 0.21	0.58 ± 0.16	0.30 ± 0.17
	U	0.96 ± 0.10	1.00 ± 0.00	0.72 ± 0.35	0.44 ± 0.19	0.31 ± 0.22	0.21 ± 0.07
+ rel + gate		1.00 ± 0.00	1.00 ± 0.00	-	-	$\mathbf{0.99 \pm 0.01}$	0.19 ± 0.04
+ abs/rel + gate		1.00 ± 0.00	1.00 ± 0.00	-	-	$\mathbf{0.98 \pm 0.02}$	$\mathbf{0.98 \pm 0.03}$
+ geo + gate (TCF)		1.00 ± 0.00	1.00 ± 0.00	-	-	$\mathbf{1.00 \pm 0.00}$	$\mathbf{1.00 \pm 0.00}$

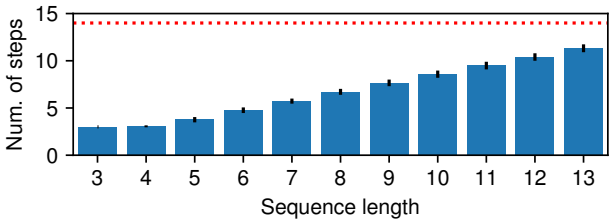

Figure 4: Average number of steps/layers for different sequence lengths on the compositional table lookup task for the Transformer with relative positional encodings and the ACT variant described in Appendix A. The red line shows $T_{\text{max}} = 14$. Note that the sequence length shown here includes the begin and end tokens. Thus, the sequence length of 4 corresponds to one function application (3 for the identity function i.e. no function is applied).

B Details of attention with absolute/relative positional encoding combination

The use of copy gates enables Transformers to generalize to longer lengths in the forward presentation order of CTL task (Sec. 3.1), but that alone was not enough to make the model to generalize in the backward order variant of the task. Examining the attention maps reveals that the model uses position-based attention for reading out the result, as opposed to content-based attention. In the backward presentation order, the last column of the transformer should focus on the second column, whose relative position changes dynamically with the length of the sequence. We solve this issue by adding an option to the attention head to choose between absolute and relative positional encodings.

In what follows, we describe the operation within a single layer/step. This allows us to omit the layer/step index t for better readability, and thus denote the state at column/position i as $\boldsymbol{h}_i$ instead of $\boldsymbol{h}^{(i,t)}$. We use the relative positional embedding variant of self-attention Dai et al. (2019). Our attention matrix with the gated absolute/relative positional encodings can be decomposed as follows:

$$r_i = \sigma(\boldsymbol{h}_i \boldsymbol{W}_{ar} + b_{ar}) \tag{16}$$

$$\hat{\boldsymbol{A}}_{i,j} = \underbrace{\boldsymbol{h}_i^\top \boldsymbol{W}_q^\top \boldsymbol{W}_{k,E} \boldsymbol{h}_j}_{(a)} + \underbrace{\boldsymbol{u}^\top \boldsymbol{W}_{k,E} \boldsymbol{h}_j}_{(c)} + \Big(\underbrace{\boldsymbol{h}_i^\top \boldsymbol{W}_q^\top \boldsymbol{W}_{k,P}}_{(b)} + \underbrace{\boldsymbol{v}^\top \boldsymbol{W}_{k,P}}_{(d)} \Big) \underbrace{\big(\boldsymbol{p}_{i-j} r_i + \boldsymbol{p}_j (1 - r_i) \big)}_{(e)} \tag{17}$$

where the matrix $\boldsymbol{W}_q \in \mathbb{R}^{d_{\text{head}} \times d}$ maps the states to queries, $\boldsymbol{W}_{k,E} \in \mathbb{R}^{d_{\text{head}} \times d}$ maps states to keys, while $\boldsymbol{W}_{k,P} \in \mathbb{R}^{d_{\text{head}} \times d}$ maps positional embedding to keys. d_{head} is the size of the key, query and value vectors for each head, set as $d_{\text{head}} = \frac{d}{n_{\text{head}}}$. $\boldsymbol{u}, \boldsymbol{v} \in \mathbb{R}^{d_{\text{head}}}$ are learned vectors. $\boldsymbol{p}_i \in \mathbb{R}^d$ is the standard sinusoidal embedding for position i (Vaswani et al., 2017). Softmax is applied to the second dimension of $\hat{\boldsymbol{A}}$ to obtain the final attention scores, $\boldsymbol{A}$. Component (a) corresponds to content-based addressing, (b, e) to content based positional addressing, (c) represents a global content bias, while (d, e) represents a global position bias.

We introduce the term (e) for the positional embedding which can switch between absolute and relative positional encodings using the scalar gate r_i (Eq. 16; parameterized by $\boldsymbol{W}_{ar} \in \mathbb{R}^{d \times 1}$ and $b_{ar} \in \mathbb{R}$), which is the function of the state at target position i.

C Implementation details

A PyTorch implementation of our models together with the experimental setup is available under `https://github.com/after_review`. The performance of all models are reported as mean and standard deviation over 5 different seeds.

C.1 Choosing the number of layers

In Sec. 2, we hypothesized that one of the conditions for our model to generalize is to be "sufficiently" deep such that elementary operations are learned in separate layers which would then become composable. In practice, a "sufficient" depth can be determined by the basic units of compositions implicitly defined by the dataset. The depth of the model must be at least as deep as the deepest path in the computation graph defined by these basic operations. This hypothesis was empirically validated in the ablation study presented above (Sec. A). In general, we used the following heuristics to choose the depth of the Transformers:

(length of the deepest path in the graph) $\times$ (steps per operation) $+$ a few more layers.

Determining the number of steps needed by the elementary operation is not straightforward but it can be done empirically. For example, for ListOps, as is shown in Sec. 4, it requires two steps per operation: one step in which the operands attend to the operation, followed by another one where the result is written back to the operation. For the other tasks, we found that a single step per operation was enough. Choosing more layers than needed is safe, and it can be used to determine the required number of layers, for example by looking at the gate activity. Finally, "$+$ a few more layers" are needed because one additional layer should be used to read out the final result, and one or a few more can be needed for communication between columns (e.g., to determine operator precedence).

Since parameters are shared across layers, we can optionally train models with a certain number of layers and increase the number of computational steps at test time. This allows us to train models using a depth which is "sufficient" to solve the training set, but increase it at test time to generalize to a test set requiring more computational steps. We did this for the ListOps experiment (Sec. 3.3): the model was trained with 20 layers and tested with 24. Our preliminary experiments confirmed that this practice has no performance penalty, while it speeds up training.

C.2 DATASET DETAILS

Compositional table lookup. Our implementation uses 8 symbols as input arguments and 9 randomly sampled bijective functions denoted by lowercased letters of the English alphabet. All functions are included in the train set in combination with all possible input symbols. The rest of the training set consists of random combinations of functions applied to a random symbol as an argument, up to length 5. The total size of the train set is 53,704 samples. The samples are roughly balanced such that there are similar numbers of samples for each depth. There are different validation sets: an IID, which matches the distribution of the train set, and a depth validation, which includes samples of length 6, 7 and 8. The test set consists of sequences of length 9 and 10.

Simple arithmetic. The dataset is constructed by sampling random digits (0-9) and operations + (add) and $*$ (multiply). The operations are performed modulo 10. The arguments of the operations are surrounded by parenthesis. The depth of the resulting tree is computed and rejection sampling is used to ensure the same number of samples from each depth present in the given split. The maximum length of samples is 50 tokens, sub-operations are sampled with probability 0.2. 100 K samples are used for training, 1 K for both test and validation set. The train set consists of 0-5 operations, the validation set 6 and the test set 7 operations.

ListOps. Random digits are sampled from range 0-9. Operations are sampled from the set sum-modulo (`SM`), which is a sum modulo 10, min (`MIN`), max (`MAX`) and median followed by the floor function (`MED`). The maximum number of arguments for each operation is 5. A sub-operation is sampled with probability 0.3. 1 M samples are used for training, 1 K for test and validation. The train set consists of 0-5 operations, 6 for the validation set and 7 for the test set.

For each sample we calculate a number which we call *dependency depth*. To understand what does it mean, note that the MIN and MAX operations only select one of its operands, MED selects 1 or 2. In sum, all operands are needed to perform the operation. If we construct a parse tree and prune away the branches which were *not* selected by any operation, and measure the depth of such a tree, the resulting number is the dependency depth. We ensure that the number of samples of each possible dependency depth is equalized in each split. This ensures that the deeper parts of the tree is actually contributing to the result calculation, preventing shallow heuristics, like ignoring all branches of the tree which are deeper than a certain depth, which otherwise would still have high probability of being correct.

C.3 MODEL DETAILS

We use the AdamW optimizer (Loshchilov & Hutter, 2019) for all of our models. Standard hyperparameters are listed in Tab. 7, 8 and 9. Additionally, models with gating use dropout (Hanson, 1990; Srivastava et al., 2014) applied to the content-based query and the position-query components of 0.1 for most of the models, except for non-gated Transformers on ListOps, where this value is 0.05. In the case of geometric attention, since the channels of the positional encoding does not have any redundancy, dropout is applied just to the content-query.

In the case of Transformers with the copy gate but without geometric attention, we use $\tanh$ instead of $\mathrm{LayerNorm}$ in Eq. 2.

The hyperparameters of the gateless Transformers differ significantly from the gated ones. This is because they are very hard to train to achieve good performance even on the IID set, and they needed extensive hyperparameter optimization. One might argue that fewer layers makes them less competitive on longer sequences, however, we were unable to train them to perform well even on the IID data with a comparable size. Similarly, increasing the size of the baseline Transformer models made them fail to train.

All Transformer variants have a begin (B) and end (E) token included on sequence boundaries. RNNs (LSTM and DNC) have no such tokens. All Transformers are encoder-only, and the result is read from the last column (corresponding to the end token).

The DNC has 21 memory cells, 4 read heads and an LSTM controller, and contains recently introduced improvements (Csordás & Schmidhuber, 2019).

We use gradient clipping with magnitude 5 (for CTL) or 1 (for simple arithmetic and ListOps) for all of our models.

Hyperparameters were obtained by a Bayesian hyperparameter search of Weights & Biases[1] over the systematically different (OOD) validation set for the `+abs/rel + gate` models and was reused for all other gated models. For the non-gated models we used the `+rel` variant for tuning. It was not possible to tune the baselines using only the OOD validation set, because their performance was too bad on that set. We thus used the mixture of the IID and OOD validation set to tune the hyperparameters for the baselines. Table 10 shows the range of hyperparameters used for tuning. "FF multiplier" is used to calculate d_{FF} from d_{model}.

We train all models to a fixed number of n_{iters} iterations and measure its validation performance every 1000 iterations. For each model, we select the best checkpoint according to the validation performance, and report its test accuracy.

Table 7: Hyperparameters used for different models on the Compositional table lookup task. We denote the feedforward size as d_{FF}, weight decay as "wd.", dropout as "do.". The model is trained for n_{iters} iterations.

	d_{model}	d_{FF}	n_{heads}	n_{layers}	batch s.	learning rate	wd.	do.	n_{iters}
LSTM	200	-	-	1	256	$10 * 10^{-4}$	-	0.5	20k
Bidirectional LSTM	400	-	-	1	256	$10 * 10^{-4}$	-	0.5	20k
DNC	200	-	-	1	256	$10 * 10^{-4}$	-	0.5	20k
Transformer	128	256	4	11	512	$1.5 * 10^{-4}$	0.0025	0.1	30k
+ rel	128	256	4	11	512	$1.5 * 10^{-4}$	0.0025	0.1	30k
+ rel + gate	256	512	1	14	512	$2 * 10^{-4}$	0.01	0.5	30k
+ abs/rel + gate	256	512	1	14	512	$2 * 10^{-4}$	0.01	0.5	30k
+ geom. att.	128	256	4	11	512	$1.5 * 10^{-4}$	0.0025	0.1	30k
+ geom. att. + gate	256	512	1	14	512	$1.5 * 10^{-4}$	0.01	0.5	30k

Table 8: Hyperparameters used for different models on the Simple arithmetic task. We denote the feedforward size as d_{FF}, weight decay as "wd.", dropout as "do.". The model is trained for n_{iters} iterations.

	d_{model}	d_{FF}	n_{heads}	n_{layers}	batch s.	learning rate	wd.	do.	n_{iters}
LSTM	200	-	-	2	256	$10 * 10^{-4}$	-	0.5	200k
Bidirectional LSTM	400	-	-	2	256	$10 * 10^{-4}$	-	0.5	200k
Transformer	128	256	4	11	512	$1.5 * 10^{-4}$	0.0025	0.5	200k
+ rel	128	256	4	11	512	$1.5 * 10^{-4}$	0.0025	0.5	200k
+ abs/rel + gate	256	1024	4	15	512	$1.5 * 10^{-4}$	0.01	0.5	100k
+ geom. att. + gate	256	1024	4	15	512	$1.5 * 10^{-4}$	0.01	0.5	100k

Table 9: Hyperparameters used for different models on the ListOps task. We denote the feedforward size as d_{FF}, weight decay as "wd.", dropout as "do.". The model is trained for n_{iters} iterations.

	d_{model}	d_{FF}	n_{heads}	n_{layers}	batch s.	learning rate	wd.	do.	n_{iters}
LSTM	512	-	-	4	512	$10 * 10^{-4}$	0.08	0.1	200k
Bidirectional LSTM	1024	-	-	4	512	$10 * 10^{-4}$	0.08	0.1	200k
Transformer	256	1024	16	6	512	$4 * 10^{-4}$	0.05	0.015	200k
+ rel	256	1024	16	6	512	$4 * 10^{-4}$	0.05	0.015	200k
+ abs/rel + gate	512	1024	16	20	512	$2 * 10^{-4}$	0.09	0.1	100k
+ geom. att. + gate	512	1024	16	20	512	$2 * 10^{-4}$	0.09	0.1	100k

[1] `https://wandb.ai/`

Table 10: Parameter ranges for hyperparameter tuning

Parameter	Range
learning rate	0.00005 ... 0.001
n_{layers}	4 ... 20
d_{model}	128, 256, 512
n_{heads}	2, 4, 8 , 16
weight decay	0.0 ... 0.1
dropout	0.0 ... 0.5
attention dropout	0.0 ... 0.5
FF multiplier	1, 2, 4

D ADDITIONAL ANALYSIS

D.1 COMPOSITIONAL TABLE LOOKUP

In the main text, we only had space to show the gate and attention activity of the TCF for a few timesteps. Here we show the corresponding visualization of all steps for the TCF in Figures 8 and 9, as well as the attention map for the baseline Transformer with relative positional encoding in Figure 5. We also show the `Transformer + abs/rel + gate` variant on Fig. 6 and Fig. 7. Please directly refer to the caption of the figures for the corresponding analysis. In general, the visualization for our TCF and the `abs/rel + gate` variant is interpretable unlike that of the baseline Transformer model.

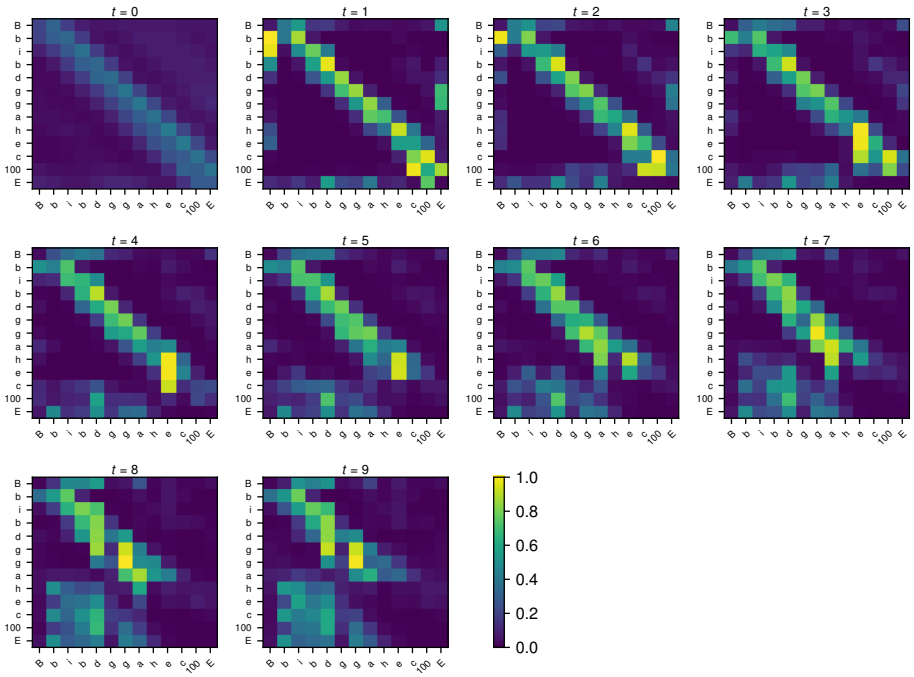

Figure 5: Attention map for every computation step for a baseline Transformer with relative positional encoding on CTL. The attention pattern gets blurry very quickly for this baseline model which does not generalize to longer sequences.

D.2 LISTOPS

Figures 10 and 12 shows the attention and gate patterns of our TCF architecture on an example from the ListOps dataset. We highlighted the notable attention patterns in Sec. 4.

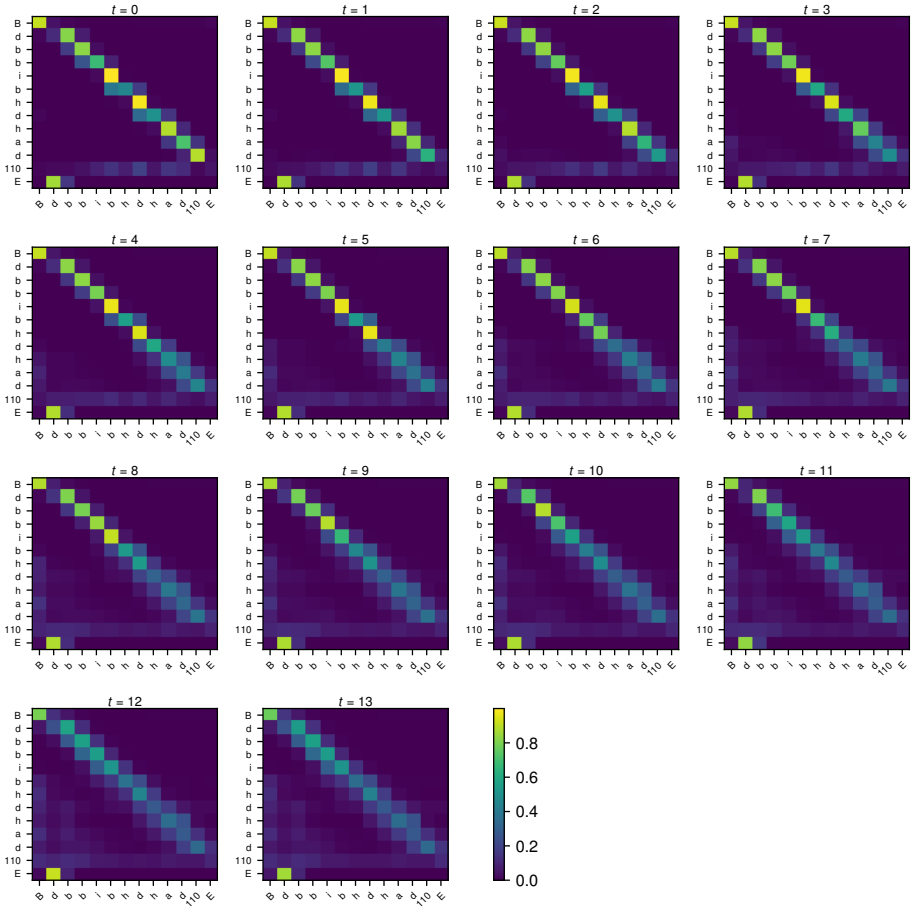

Figure 6: Attention map for every computation step for a Transformer with gating and relative/absolute positional encoding (presented in Figure 2) on CTL. The attention pattern is relatively stable over time, and its gets blurrier only after the given column is processed and updated. The gate sequence for the same input can be seen on Figure 7

Different heads seem to specialize in different functions. As has been already commented in Sec. 4, *head 13* of the TCF architecture, shown in Figure 11, seems to specialize in selecting which which arguments belong to which operator.

The gating patterns are also very interesting. In the early stages, the deepest part of the input is updated: `[MAX 2 4 0 8 9]` and `[MED 8 5 8]`, which are independent branches of the parse tree and can be processed in parallel. In the following steps, the update patterns spread up in the parse tree, updating the operations that have its arguments available. The input is read from the first column in this case, which is written the last, in a very late stage.

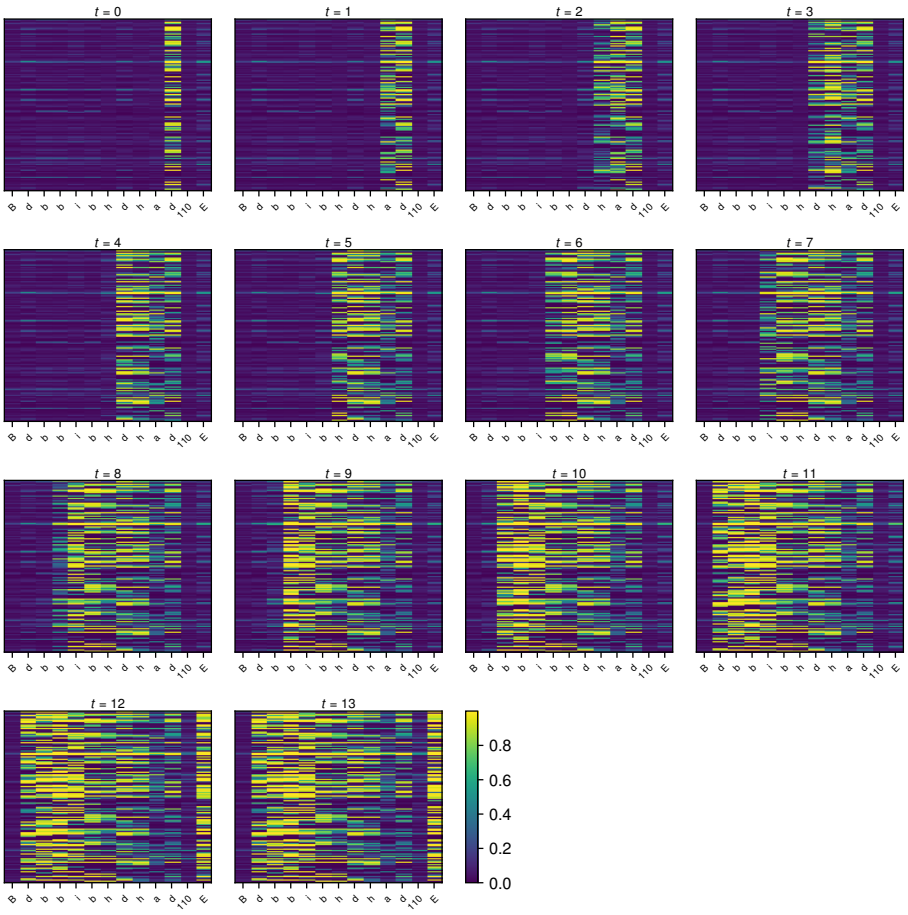

Figure 7: Gates for every computation step for a Transformer with gating and relative/absolute positional encoding on CTL. The gates are closed until all arguments of the given operation become available. The attention maps for the same input can be seen on Figure 6.

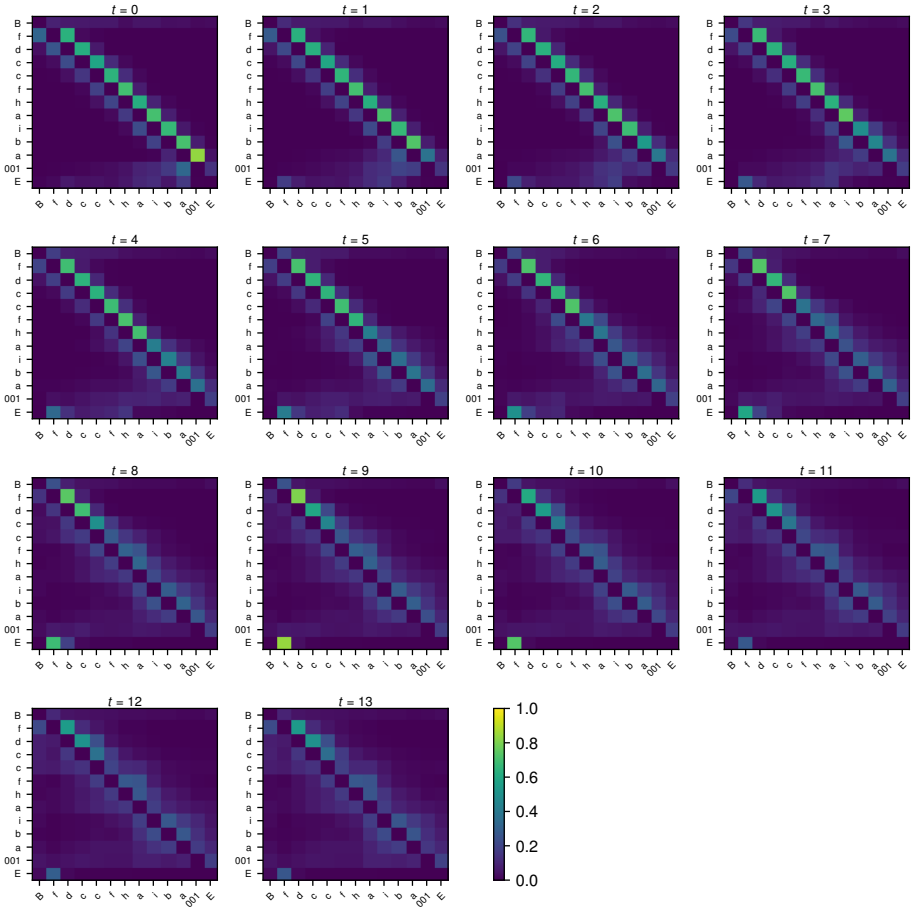

Figure 8: Attention map for every computation step of the TCF on CTL. It can be seen that the network correctly and clearly focuses on the last element of the sequence and the last sharp read happens in step 10 - corresponding to the 10 function calls in the example. The gate sequence for the same input can be seen in Figure 9

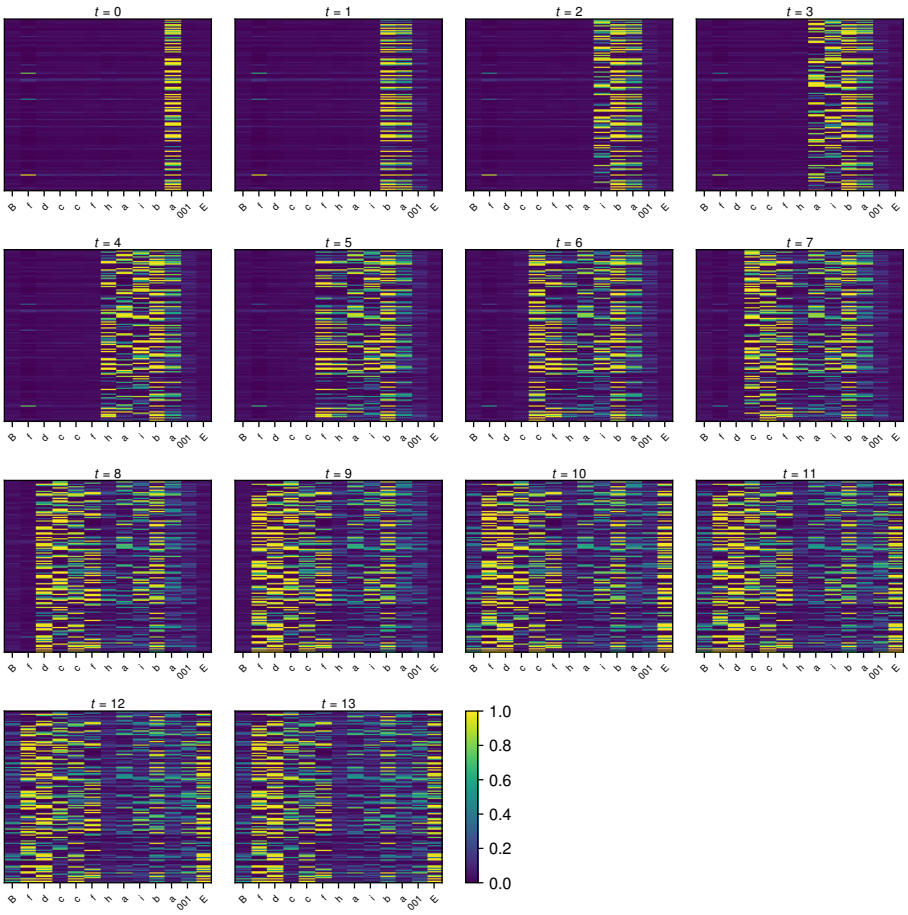

Figure 9: Gates for every computation step of the TCF on CTL. The gates remain closed until all arguments of the given operations become available. The attention maps for the same input can be seen in Figure 8.

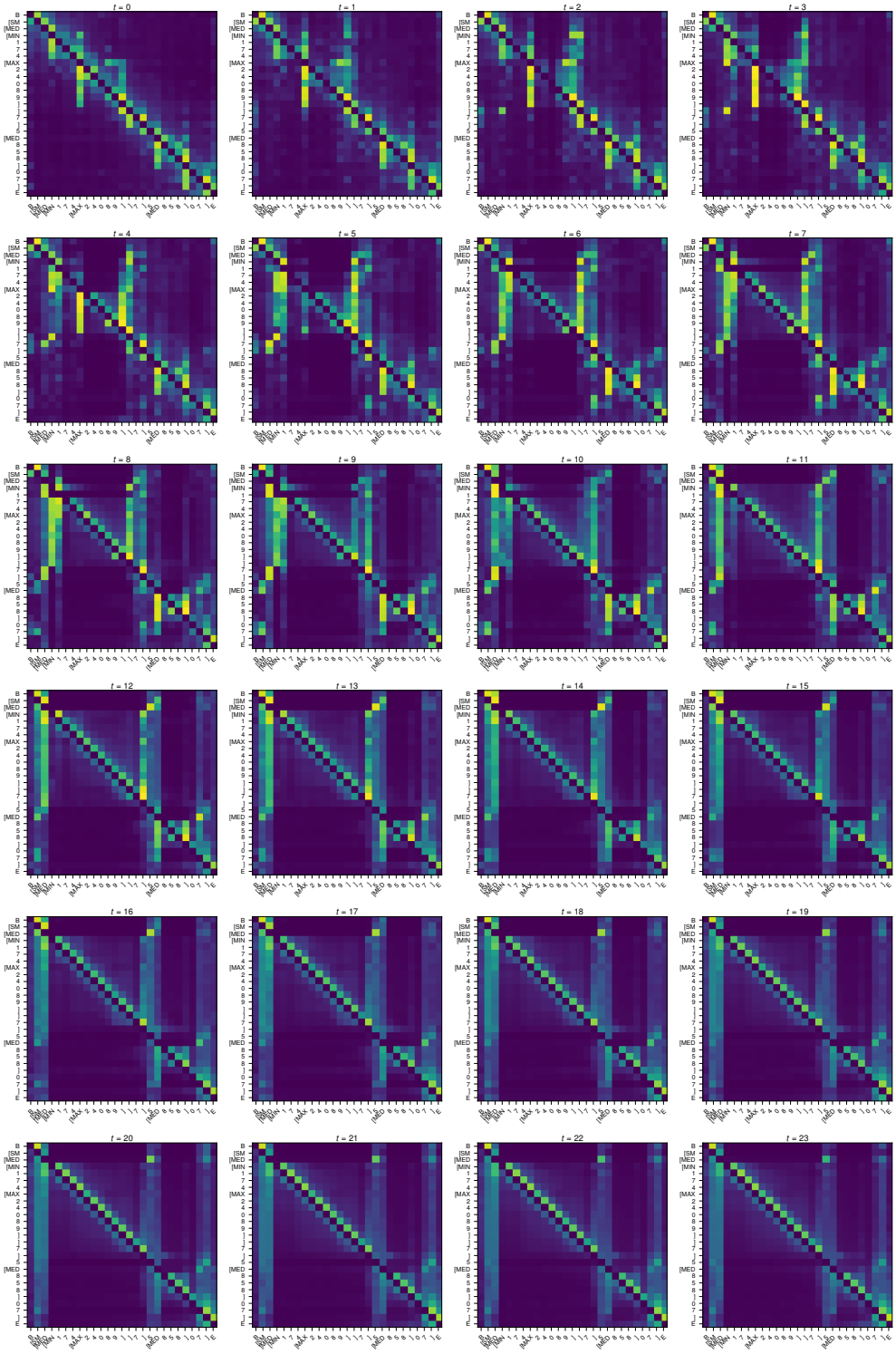

Figure 10: Attention maps for every computation step of the TCF on ListOps. The network has 16 heads, the max of them is shown. The input is only depth 4, which explains the computation stopping early, rougly after 8-9 steps, after which the attention barely changes. The corresponding gate maps for the same input can be seen in Figure 12.

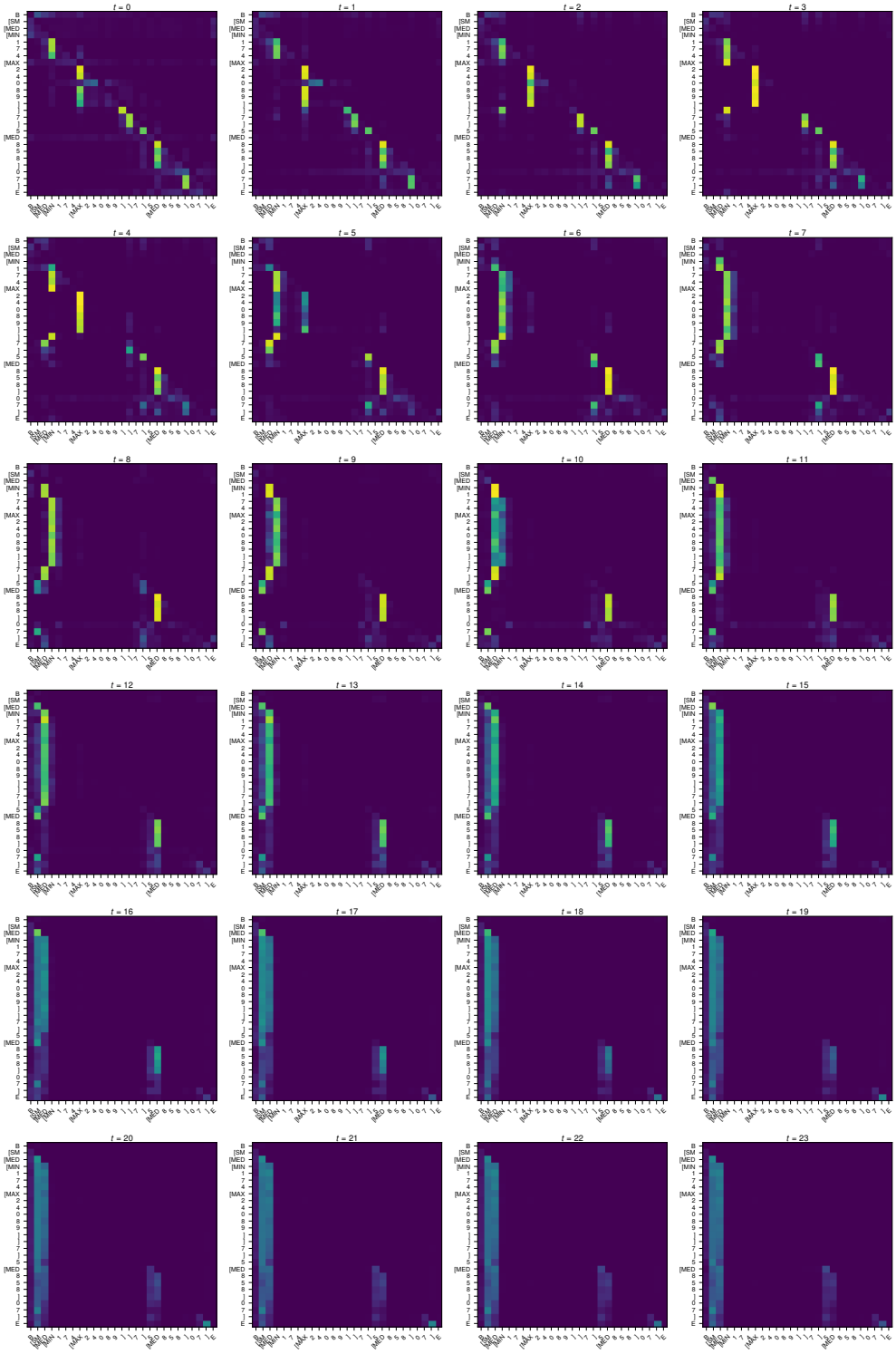

Figure 11: Attention maps for *head 13* of the TCF in every computation step on ListOps. This head shows the operands for each operation. Following this head, we can observe the hierarchy and the order in which the operations are performed.

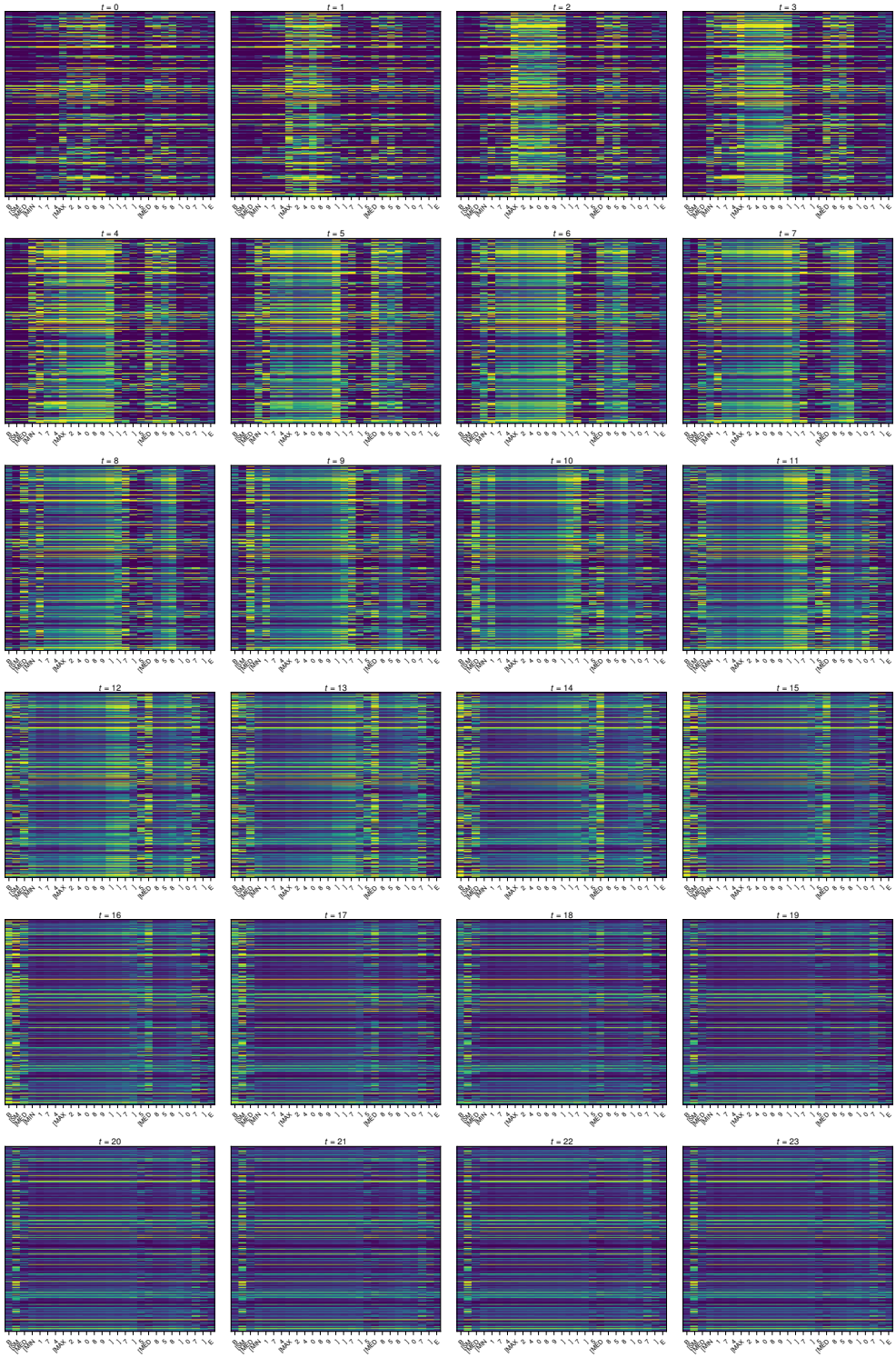

Figure 12: Gates for every computation step of the TCF on ListOps. It can be seen that gates open from the deepest operations in the tree and process upwards in the computation tree. The input is only depth 4, which explains the computation stopping early, rougly after 8-9 steps. The attention maps for the same input can be seen in Figure 10.