On the Ability and Limitations of Transformers to Recognize Formal Languages

Satwik Bhattamishra[♠] **Kabir Ahuja**[♢*] **Navin Goyal**[♠]
[♠] Microsoft Research India
[♢] Udaan.com
{t-satbh,navingo}@microsoft.com
kabir.ahuja@udaan.com

Abstract

Transformers have supplanted recurrent models in a large number of NLP tasks. However, the differences in their abilities to model different syntactic properties remain largely unknown. Past works suggest that LSTMs generalize very well on regular languages and have close connections with counter languages. In this work, we systematically study the ability of Transformers to model such languages as well as the role of its individual components in doing so. We first provide a construction of Transformers for a subclass of counter languages, including well-studied languages such as n-ary Boolean Expressions, Dyck-1, and its generalizations. In experiments, we find that Transformers do well on this subclass, and their learned mechanism strongly correlates with our construction. Perhaps surprisingly, in contrast to LSTMs, Transformers do well only on a subset of regular languages with degrading performance as we make languages more complex according to a well-known measure of complexity. Our analysis also provides insights on the role of self-attention mechanism in modeling certain behaviors and the influence of positional encoding schemes on the learning and generalization abilities of the model.

1 Introduction

Transformer (Vaswani et al., 2017) is a self-attention based architecture which has led to state-of-the-art results across various NLP tasks (Devlin et al., 2019; Liu et al., 2019; Radford et al., 2018). Much effort has been devoted to understand the inner workings and intermediate representations of pre-trained models; Rogers et al. (2020) is a recent survey. However, our understanding of their practical ability to model different behaviors relevant to sequence modeling is still nascent.

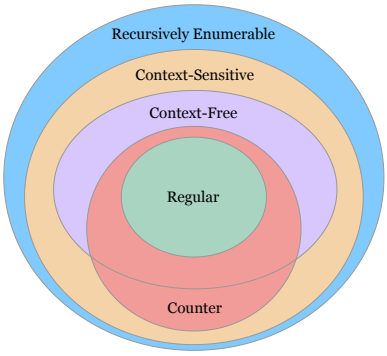


Figure 1: Counter languages form a strict superset of regular languages, and are a strict subset of context-sensitive languages. Counter and context-free languages have a nonempty intersection and neither set is contained in the other.

On the other hand, a long line of research has sought to understand the capabilities of recurrent neural models such as the LSTMs (Hochreiter and Schmidhuber, 1997) . Recently, Weiss et al. (2018), Suzgun et al. (2019a) showed that LSTMs are capable of recognizing counter languages such as Dyck-1 and $a^n b^n$ by learning to perform counting like behavior. Suzgun et al. (2019a) showed that LSTMs can recognize shuffles of multiple Dyck-1 languages, also known as Shuffle-Dyck. Since Transformer based models (e.g., GPT-2 and BERT) are not equipped with recurrence and start computation from scratch at each step, they are incapable of directly maintaining a counter. Moreover, it is known that theoretically RNNs can recognize any regular language in finite precision, and LSTMs work well for this task in practical settings. However, Transformer's ability to model such properties in practical settings remains an open question.

Prior to the current dominance of Transformers for NLP tasks, recurrent models like RNN-based models such as LSTMs were the most common choice, and their computational capabilities have

*This research was conducted during the author's internship at Microsoft Research.

been studied for decades, e.g., (Kolen and Kremer, 2001). In this work, we investigate the ability of Transformers to express, learn, and generalize on certain counter and regular languages. Formal languages provide us a controlled setting to study a network's ability to model different syntactic properties in isolation and the role of its individual components in doing so.

Recent work has demonstrated close connections between LSTMs and counter automata. Hence, we seek to understand the capabilities of Transformers to model languages for which the abilities of LSTMs are well understood. We first show that Transformers are expressive enough to recognize certain counter languages like Shuffle-Dyck and n-ary Boolean Expressions by using self-attention mechanism to implement the relevant counter operations in an indirect manner. We then extensively evaluate the model's learning and generalization abilities on such counter languages and find that models generalize well on such languages. Visualizing the intermediate representations of these models shows strong correlations with our proposed construction. Although Transformers can generalize well on some popularly used counter languages, we observe that they are limited in their ability to recognize others. We find a clear contrast between the performance of Transformers and LSTMs on regular languages (a subclass of counter languages). Our results indicate that, in contrast to LSTMs, Transformers achieve limited performance on languages that involve modeling periodicity, modular counting, and even simpler star-free variants of Dyck-1, which they were able to recognize effortlessly. Our analysis provides insights about the significance of different components, namely self-attention, positional encoding, and the number of layers. Our results also show that positional masking and positional encoding can both aid in generalization and training, but in different ways. We conduct extensive experiments on over 25 carefully chosen formal languages. Our results are perhaps the first indication of the limitations of Transformers for practical-sized problems that are, in a precise sense, very simple, and in particular, easy for recurrent models.

2 Related Work

Numerous works, e.g., Suzgun et al. (2019b); Sennhauser and Berwick (2018); Skachkova et al. (2018), have attempted to understand the capabilities and inner workings of recurrent models by empirically analyzing them on formal languages. Weiss et al. (2018) showed that LSTMs are capable of simulating counter operations and explored their practical ability to recognize languages like $a^n b^n$ and $a^n b^n c^n$. Suzgun et al. (2019a) further showed that LSTMs can learn to recognize Dyck-1 and Shuffle-Dyck and can simulate the behavior of k-counter machines. Theoretical connections of recurrent models have been established with counter languages (Merrill, 2019; Merrill et al., 2020; Merrill, 2020). It has also been shown that RNN based models can recognize regular languages (Kolen and Kremer, 2001; Korsky and Berwick, 2019) and efforts have been made to extract DFAs from RNNs trained to recognize regular languages (Weiss et al., 2019; Wang et al., 2018b; Michalenko et al., 2019). We are not aware of such studies for Transformers.

Recently, researchers have sought to empirically analyze different aspects of the Transformer trained on practical NLP tasks such as the information contained in intermediate layers (Rogers et al., 2020; Reif et al., 2019; Warstadt et al., 2019). Voita et al. (2019) studied the role of different types of attention heads. Yang et al. (2019); Tsai et al. (2019) examined the ability of the model to learn order information via different positional encoding schemes. Complementary to these, our work is focused on analyzing Transformer's ability to model particular behaviors that could be relevant to modeling linguistic structure. Recently, it has been shown that Transformers are Turing-complete (Pérez et al., 2019; Bhattamishra et al., 2020) and are universal approximators of sequence-to-sequence functions given arbitrary precision (Yun et al., 2020). Hahn (2020) shows that Transformers cannot recognize languages Parity and Dyck-2. However, these results only apply to very long words, and their applicability to practical-sized inputs is not clear (indeed, we will see different behavior for practical-sized input). Moreover, these results concern the expressive power of Transformers and do not apply to learning and generalization abilities. Thus Transformers' ability to model formal languages requires further investigation.

3 Definitions

We consider the Transformer as used in popular pre-trained LM models such as BERT and GPT, which is the encoder-only model of the original seq-to-seq architecture (Vaswani et al., 2017). The encoder

consists of multiple layers with two blocks each: (1) self-attention block, (2) a feed-forward network (FFN). For $1 \leq i \leq n$, at the i-th step, the model takes as input the sequence $s_1, s_2, \ldots, s_i$ where $s \in \Sigma$ and generates the output vector $\boldsymbol{y}_i$. Each input s_i is first converted into an embedding vector using the function $f_e : \Sigma \to \mathbb{R}^{d_{\text{model}}}$ and usually some form of positional encoding is added to yield the final input vector $\boldsymbol{x}_i$. The embedding dimension d_{model} is also the dimension of intermediate vectors of the network. Let $\boldsymbol{X}_i := (\boldsymbol{x}_1, \ldots, \boldsymbol{x}_i)$ for $i \geq 1$.

In the self-attention block, the input vectors undergo linear transformations $Q(\cdot), K(\cdot),$ and $V(\cdot)$ yielding the corresponding query, key and value vectors, respectively. The self-attention mechanism takes as input a *query* vector $Q(\boldsymbol{x}_i)$, *key* vectors $K(\boldsymbol{X}_i)$, and *value* vectors $V(\boldsymbol{X}_i)$. An attention-head denoted by $\text{Att}(Q(\boldsymbol{x}_i), K(\boldsymbol{X}_i), V(\boldsymbol{X}_i))$, is a vector $\boldsymbol{a}_i = \sum_{j=1}^{i} \alpha_j \boldsymbol{v}_j$, where $(\alpha_1, \ldots, \alpha_i) = \text{softmax}(\langle Q(\boldsymbol{x}_i), K(\boldsymbol{x}_1)\rangle, \ldots, \langle Q(\boldsymbol{x}_i), K(\boldsymbol{x}_i)\rangle)$.

The output of a layer denoted by $\boldsymbol{z}_i$ is computed by $\boldsymbol{z}_i = O(\boldsymbol{a}_i)$ where $1 \leq i \leq n$ and $O(\cdot)$ typically denotes an FFN with ReLU activation. The complete L-layer model is a repeated application of the single-layer model described above, which produces a vector $\boldsymbol{z}_i^L$ at its final layer where L denotes the last layer. The final output is obtained by applying a projection layer with some normalization or an FFN over the vectors $\boldsymbol{z}_i^L$'s and is denoted by $\boldsymbol{y}_i = F(\boldsymbol{z}_i^L)$. Residual connections and layer normalization are also applied to aid the learning process of the network.

In an LM setting, when the Transformer processes the input sequentially, each input symbol can only attend over itself and the previous inputs, masking is applied over the inputs following it. Note that, providing positional information in this form via masked self-attention is also referred to as positional masking (Vaswani et al., 2017; Shen et al., 2018). A Transformer model without positional encoding and positional masking is order-insensitive.

3.1 Formal Languages

Formal languages are abstract models of the syntax of programming and natural languages; they also relate to cognitive linguistics, e.g., Jäger and Rogers (2012); Hahn (2020) and references therein.

Counter Languages. These are languages recognized by a deterministic counter automaton (DCA), that is, a DFA with a finite number of unbounded counters (Fischer et al., 1968). The counters can be incremented/decremented by constant values and can be reset to 0 (details in App. B.1). The commonly used counter languages to study sequence models are Dyck-1, $a^n b^n$, and $a^n b^n c^n$. Several works have explored the ability of recurrent models to recognize these languages as well as their underlying mechanism to do so. We include them in our analysis as well as some general form of counter languages such as Shuffle-Dyck (as used in Suzgun et al. (2019a)) and n-ary Boolean Expressions. The language Dyck-1 over alphabet $\Sigma = \{[,]\}$ consists of balanced parentheses defined by derivation rules $S \to [\ S\ ] \mid SS \mid \epsilon$. Shuffle-Dyck is a family of languages containing shuffles of Dyck-1 languages. Shuffle-k denotes the shuffle of k Dyck-1 languages: it contains k different types of brackets, where each type of bracket is required to be well-balanced, but their relative order is unconstrained. For instance, a Shuffle-2 language over alphabet $\Sigma = \{[,], (,)\}$ contains the words $([])$ and $[(()])$ but not $])[($. We also consider n-ary Boolean Expressions (hereby BoolExp-n), which are a family of languages of valid Boolean expressions (in the prefix notation) parameterized by the number of operators and their individual arities. For instance, an expression with unary operator $\sim$ and binary operator $\wedge$ contains the word '$\wedge \sim 01$' but not '~ 10' (formal definitions in App. B).

Note that, although languages such as Dyck-1 and $a^n b^n$ are context-free, a DCA with a single counter is sufficient to recognize Dyck-1 and $a^n b^n$. Similarly, a DCA with two single-turn counters can recognize $a^n b^n c^n$. On the other hand, recognizing Shuffle-Dyck requires multiple multi-turn counters, where for a given type of bracket, its corresponding counter is incremented or decremented by 1. Hence, it represents a more general form of counter languages. Similarly, recognizing BoolExp requires a 1-counter DCA with the counter updates depending on the operator: a ternary operator will increment the counter by 2 ($= \text{arity} - 1$) whereas a unary operator will increment it by 0. Figure 1 shows the relationship between counter languages and other classes of formal languages.

Regular Languages. Regular languages, perhaps the best studied class of formal languages, form a subclass of counter languages[1]. They neatly divide

[1]For simplicity, from now on, we will refer to a particular language as a counter language if a DCA with a nonzero number of counters is necessary to recognize it, else we will refer to it as a regular language.

into two subclasses: *star-free* and *non-star-free*. Star-free languages can be described by regular expressions formed by union, intersection, complementation, and concatenation operators but not the Kleene star ($*$). Like regular languages, star-free languages are surprisingly rich with algebraic, logical, and multiple other characterizations and continue to be actively researched, e.g., (McNaughton and Papert, 1971; Jäger and Rogers, 2012). They form a simpler subclass of regular languages where the notion of simplicity can be made precise in various ways, e.g. they are first-order logic definable and cannot represent languages that require modular counting.

We first consider Tomita grammars containing 7 regular languages representable by DFAs of small sizes, a popular benchmark for evaluating recurrent models and extracting DFA from trained recurrent models (see, e.g., Wang et al. (2018a)). Tomita grammars contain both star-free and non-star-free languages. We further investigate some non-star-free languages such as $(aa)^*$, Parity and $(abab)^*$. Parity contains words over $\{0, 1\}$ with an even number of 1's. Similarly $(aa)^*$ and $(abab)^*$ require modeling periodicity.

On the other hand, the seemingly similar looking language $(ab)^*$ is star-free: $(ab)^* = (b\emptyset^c + \emptyset^c a + \emptyset^c aa\emptyset^c + \emptyset^c bb\emptyset^c)^c$, where $\cdot^c$ denotes set complementation, and thus $\emptyset^c = \Sigma^*$. The dot-depth of a star-free language is a measure of nested concatenation or sequentiality required in a star-free regular expression (formal definition in App. B.2). We define a family $\mathcal{D}_0, \mathcal{D}_1, \ldots$ of star-free languages. For $n \geq 0$, the language $\mathcal{D}_n$ over $\Sigma = \{a, b\}$ is defined inductively as follows: $\mathcal{D}_n = (a\mathcal{D}_{n-1}b)^*$ where $\mathcal{D}_0 = \epsilon$, the empty word. Thus $\mathcal{D}_1 = (ab)^*$ and $\mathcal{D}_2 = (a(ab)^*b)^*$. Language $\mathcal{D}_n$ is known to have dot-depth n.

The list of all considered languages and their definitions are provided in the App. B.

4 Expressiveness Results

Proposition 4.1. *There exists a Transformer as defined in Section 3 that can recognize the family of languages Shuffle-Dyck.*

Proof. Let $s_1, s_2, \ldots, s_n$ denote a sequence $w \in$ Shuffle-k over the alphabet $\Sigma = \{[_0, \ldots, [_{k-1},]_0, \ldots,]_{k-1}\}$. The language Shuffle-1 is equivalent to Dyck-1. For any Shuffle-k language, consider a model with $d_{model} = 2k$, where the embedding function f_e is defined as follows. For each type of open bracket $[_j$ where,$0 \leq j < k$, the vector $f_e([_j)$ has the value $+1$ and -1 at the indices $2j$ and $2j+1$, respectively. It has the value 0 at the rest of the indices. Similarly for each closing bracket, the vector $f_e(]_j)$ has the value -1 and $+1$ at the indices $2j$ and $2j+1$, and it has the value 0 at the rest of the indices. For Dyck-1, this would lead to $f_e([) = [+1, -1]^T$ and $f_e(]) = [-1, +1]^T$ (with $d_{model} = 2$). We use a single-layer Transformer where we set the matrix corresponding to linear transformation for key vectors to be null matrix, that is $K(\boldsymbol{x}) = \mathbf{0}$ for all $\boldsymbol{x}$. This will lead to equal attention weights for all inputs. The matrices corresponding to $Q(\cdot)$ and $V(\cdot)$ are set to Identity. Thus, $\mathrm{Att}(Q(\boldsymbol{x}_i), K(\boldsymbol{X}_i), V(\boldsymbol{X}_i)) = \frac{1}{i}\sum_{j=1}^{i} \boldsymbol{v}_j$ for $1 \leq i \leq n$. Hence, at the i-th step, the self-attention block produces a vector $\boldsymbol{a}_i$ which has the values $\frac{\sigma([_j) - \sigma(]_j)}{i}$ at indices $2j$ and the values $\frac{\sigma(]_j) - \sigma([_j)}{i}$ at indices $2j+1$, where $\sigma(s)$ denotes the number of occurrence of the symbol s. For instance, in Dyck-1, if in the first i inputs, there are $\sigma([)$ open brackets and $\sigma(])$ closing brackets, then $\boldsymbol{a}_i = [\frac{\sigma([) - \sigma(])}{i}, \frac{\sigma(]) - \sigma([)}{i}]^T$, where $i = \sigma([) + \sigma(])$. In $\boldsymbol{a}_i$, the value $\sigma([) - \sigma(])$ represents the depth (difference between the number of open and closing brackets) of the Dyck-1 word at index i. Hence, the first coordinate is the ratio of the depth of the Dyck-1 word and its length at that index, while the other coordinate is its negative.

We then apply a simple FFN with ReLU activation over the vector $\boldsymbol{a}_i$. The vector $\boldsymbol{z}_i = \mathsf{ReLU}(\mathbf{I}\boldsymbol{a}_i)$. The even indices of the vector $\boldsymbol{z}_i$ will be nonzero if the number of open brackets of the corresponding type is greater than the number of closing brackets. A similar statement holds for the odd indices. Thus, for a given word to be in Shuffle-k, the values at odd indices of the vector $\boldsymbol{z}_i$ must never be nonzero, and the values of all coordinates must be zero at the last step to ensure the number of open and closing brackets are the same.

For an input sequence $s_1, s_2, \ldots, s_n$, the model will produce $\boldsymbol{z}_1, \ldots, \boldsymbol{z}_n$ based on the construction specified above. A word w belongs to language Shuffle-k if $\boldsymbol{z}_{i,2j+1} = 0$ for all $1 \leq i \leq n, 0 \leq j < k$ and $\boldsymbol{z}_n = \mathbf{0}$ and does not belong to the language otherwise. This can be easily implemented by an additional layer of self-attention and feedforward network to classify a given sequence.

□

The bottleneck for precision in the construc-

Language	Model	Bin-1 Accuracy [1, 50]↑	Bin-2 Accuracy [51, 100]↑	Bin-3 Accuracy [101, 150]↑
Shuffle-2	**LSTM (Baseline)**	**100.0**	**100.0**	**100.0**
	Transformer (Absolute Positional Encodings)	**100.0**	85.2	63.3
	Transformer (Relative Positional Encodings)	**100.0**	51.6	3.8
	Transformer (Only Positional Masking)	**100.0**	**100.0**	93.0
BoolExp-3	**LSTM (Baseline)**	**100.0**	**100.0**	99.7
	Transformer (Absolute Positional Encodings)	**100.0**	90.6	51.3
	Transformer (Relative Positional Encodings)	**100.0**	96.0	68.4
	Transformer (Only Positional Masking)	**100.0**	**100.0**	**99.8**
$a^n b^n c^n$	**LSTM (Baseline)**	**100.0**	**100.0**	97.8
	Transformer (Absolute Positional Encodings)	**100.0**	62.1	5.3
	Transformer (Relative Positional Encodings)	**100.0**	31.3	22.0
	Transformer (Only Positional Masking)	**100.0**	**100.0**	**100.0**

Table 1: The performance of Transformers and LSTMs on the respective counter languages. Refer to section 6 for details. Performance on other counter languages such as Shuffle-4 and Shuffle-6 are listed in Table 8 in appendix.

tion above is the calculation of values of the form $\frac{\sigma([]-\sigma([])}{i}$ in the vector $\boldsymbol{a}_i$. Since in a finite precision setting with r bits, this can be computed up to a value exponential in r, our proof entails that Transformers can recognize languages in Shuffle-Dyck for lengths exponential in the number of bits.

Using a similar logic, one can also show that Transformers can recognize the family of languages BoolExp-n (refer to Lemma C.2). By setting the value vectors according to the arities of the operators, the model can obtain the ratio of the counter value of the underlying automata and the length of the input at each step via self-attention. Although the above construction is specific to these language families, we provide a proof for a more general but restricted subclass of Counter Languages in the appendix (refer to Lemma C.1). The above construction serves to illustrate how Transformers can recognize such languages by indirectly doing relevant computations. As we will later see, this will also help us interpret how trained models recognize such languages.

5 Experimental Setup

In our experiments, we consider 27 formal languages belonging to different parts in the hierarchy of counter and regular languages. For each language, we generate samples within a fixed-length window for our training set and generate multiple validation sets with different windows of length to evaluate the model's generalization ability.

For most of the languages, we generate 10k samples for our training sets within lengths 1 to 50 and create different validation sets containing samples with distinct but contiguous windows of length. The number of samples in each validation set is 2k, and the width of each window is about 50. For languages that have very few positive examples in a given window of length, such as $(ab)^*$ and $a^n b^n c^n$, we train on all positive examples within the training window. Similarly, each validation set contains all possible strings of the language for a particular range. Table 6 in appendix lists the dataset statistics of all 27 formal languages we consider.[2]. We have made our source code available at https://github.com/satwik77/Transformer-Formal-Languages.

5.1 Training details

We train the model on character prediction task as introduced in Gers and Schmidhuber (2001) and as used in Suzgun et al. (2019b,a). Similar to an LM setup, the model is only presented with positive samples from the given language. For an input sequence $s_1, s_2, \ldots, s_n$, the model receives the sequence $s_1, \ldots, s_i$ for $1 \leq i \leq n$ at each step i and the goal of the model is to predict the next set of legal/valid characters in the $(i+1)^{th}$ step. From here onwards, we say a model can recognize a language if it can perform the character prediction task perfectly.

The model assigns a probability to each character in the vocabulary of the language corresponding to its validity in the next time-step. The output can be represented by a k-hot vector where each coordinate corresponds to a character in the vocabulary of the language. The output is computed by applying a sigmoid activation over the scores assigned by the model for each character. Following Suzgun et al. (2019b,a), the learning objective of

[2]Our experimental setup closely follows the setup of Suzgun et al. (2019a,b) for RNNs

the model is to minimize the mean-squared error between the predicted probabilities and the k-hot labels.[3] During inference, we use a threshold of 0.5 to obtain the predictions of the model. For a test sample, the model's prediction is considered to be correct if and only if its output at every step is correct. Note that, this is a relatively stringent metric as a correct prediction is obtained only when the output is correct at every step. The accuracy of the model over test samples is the fraction of total samples predicted correctly[4]. Similar to Suzgun et al. (2019a) we consider models of small sizes to prevent them from memorizing the training set and make it feasible to visualize the model. In our experiments, we consider Transformers with up to 4 layers, 4 heads and the dimension of the intermediate vectors within 2 to 32. We extensively tune the model across various hyperparameter settings. We also examine the influence of providing positional information in different ways such as absolute encodings, relative encodings (Dai et al., 2019) and using only positional masking without any explicit encodings.

6 Results on Counter Languages

We evaluated the performance of the model on 9 counter languages. Table 1 shows the performance of different models described above on some representative languages. We also include the performance of LSTMs as a baseline. We found that Transformers of small size (single head and single layer) can generalize well on some general form of counter languages such as Shuffle-Dyck and BoolExp-n. Surprisingly, we observed this behavior when the network was not provided any form of explicit positional encodings, and positional information was only available in the form of masking. For models with positional encoding, the lack of the ability to generalize to higher lengths could be attributed to the fact that the model has never been trained on some of the positional encodings that it receives at test time. On the other hand, the model without any explicit form of positional encoding is less susceptible to such issues if it is capable of performing the task and was found to generalize well across various hyperparameter settings.

[3]We also tried BCE loss in our initial experiments and found similar results for languages such as Parity, Tomita grammars and certain counter languages.

[4]A discussion on the choice of character prediction task and its relation to other tasks such as standard classification and LM is provided in section D.1 in the appendix.

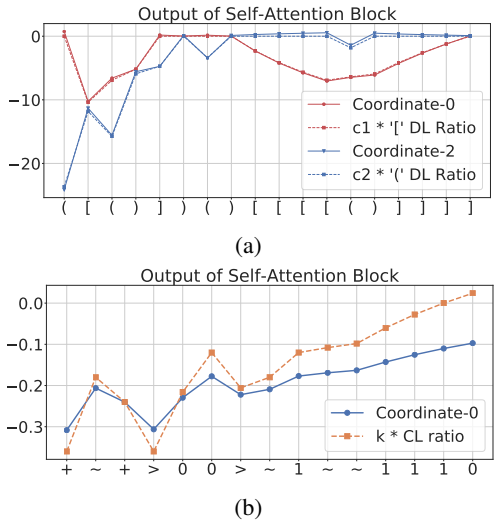


Figure 2: Values of different coordinates of the output of self-attention block of the models trained on Shuffle-2 and BoolExp-3. The dotted lines are the scaled depth to length ratios for Shuffle-2 and scaled counter value to length ratios for BoolExp-3. We observe a near perfect Pearson correlation coefficent of 0.99 between outputs of self attention block and the DL and CL ratios.

6.1 Role of Self-Attention

In order to check our hypothesis in Sec. 4, we visualize certain attributes of trained models that generalize well on Shuffle-2 and BoolExp-3.[5] Our construction in Sec. 4 recognizes sequences in Shuffle-Dyck by computing the depth to length ratio of the input at each step via self-attention mechanism. For BoolExp-n, the model can achieve the task similarly by computing the corresponding counter value divided by length (refer to Lemma C.2). Interestingly, upon visualizing the outputs of the self-attention block for a model trained on Shuffle-2, we found a strong correlation of its elements with the depth to length ratio. As shown in Fig. 2a, different coordinates of the output vector of the self-attention block contain computations corresponding to different counters of the Shuffle-2 language. We observe the same behavior for models trained on Shuffle-4 language (refer to Figure 5 in appendix). Similarly, upon visualizing a model trained on Boolean Expressions with 3 operators, we found strong correlation[6] between its elements and the ratio of the counter value and length of the input (refer to Figure 2b). This indicates that the model learns to recognize inputs by carrying out the required computation in an indirect manner,

[5]We take the model with the smallest number of parameters that generalized well making it feasible for us to visualize it.

[6]The Pearson correlation of values were ~ 0.99

Model Type	1-Layer		2-Layer	
	Bin 0	Bin 1	Bin 0	Bin 1
Positional Masking	45.1	38.9	100.0	99.2
Positional Encoding	55.8	37.9	100.0	99.6
LSTM	100.0	100.0	100.0	100.0

Table 2: Results on language Reset-Dyck-1 with different number of layers.

Language	Star-Free	Transformer		LSTM	
		Bin 0	Bin 1	Bin 0	Bin 1
Tomita 1	✓	100.0	100.0	100.0	100.0
Tomita 2	✓	100.0	100.0	100.0	100.0
Tomita 3	✗	75.4	10.8	100.0	100.0
Tomita 4	✓	100.0	92.4	100.0	100.0
Tomita 5	✗	29.3	0.0	100.0	100.0
Tomita 6	✗	88.8	0.0	100.0	100.0
Tomita 7	✓	100.0	100.0	100.0	100.0

Table 3: Results on Tomita grammar

as described in our construction. Additionally, for both models, we found that the attention weights of the self-attention block were uniformly distributed (refer to Figure 4 in appendix). Further, on inspecting the embedding and value vectors of the open and closing brackets, we found that their respective coordinates were opposite in sign and similar in magnitude. As opposed to Shuffle-Dyck, for BoolExp-n, the magnitudes of the elements in the value vectors were according to their corresponding arity. For instance, the magnitude for a ternary operator was (almost) thrice the magnitude for a unary operator (refer to Figure 3 in appendix). These observations are consistent with our construction, indicating that the model uses its value vectors to determine the counter updates and then at each step, aggregates all the values to obtain a form of the final counter value in an indirect manner. This is complementary to LSTMs, which can simulate the behavior of k-counters more directly by making respective updates to its cell states upon receiving each input (Suzgun et al., 2019a).

6.2 Limitations of the Single-Layer Transformer

Although we observed that single-layer Transformers are easily able to recognize some of the popularly studied counter languages, at the same time, it is not necessarily true for counter languages that require reset operations. We define a variant of the Dyck-1 language. Let Reset-Dyck-1 be the language defined over the alphabet $\Sigma = \{[,], \#\}$, where $\#$ denotes a symbol that resets the counter. Words in Reset-Dyck-1 have the form $\Sigma^*\#v$, where the string v belongs to Dyck-1. When the machine encounters the reset symbol $\#$, it must ignore all the previous input, reset the counter to 0 and go to start state. It is easy to show that this cannot be directly implemented with a single layer self-attention network with positional masking (Lemma C.3 in Appendix). The key limitation for both with and without encodings is the fact that for a single layer network the scoring function $\langle Q(\boldsymbol{x}_n), K(\boldsymbol{x}_\#)\rangle$ and the value vector corresponding to the reset symbol is independent of the preceding inputs which it is supposed to negate (reset). The same limitation does not hold for multi-layer networks where the value vector, as well as the scoring function for the reset symbol, are dependent on its preceding inputs. On evaluating the model on data generated from such a language, we found that single-layer networks are unable to perform well in contrast to networks with two layers (Table 2)[7]. LSTMs, on the other hand, can emulate the reset operation using forget gate.

7 Results on Regular Languages

We first examine the popular benchmark of Tomita grammars. While the LSTMs generalize perfectly on all 7 languages, Transformers are unable to generalize on 3 languages, all of which are non-star-free. Note that, all star-free languages in Tomita grammar have dot-depth 1. Recognizing non-star-free languages requires modeling properties such as periodicity and modular counting. Consequently, we evaluate the model on some of the simplest non-star-free languages such as the languages $(aa)^*$ and Parity. We find that they consistently fail to learn or generalize on such languages, whereas LSTMs of very small sizes perform flawlessly. Table 4 lists the performance on some non-star-free languages. Note that LSTMs can easily recognize such simple non-star-free languages considered here by using its internal memory and recurrence [8]. However, doing the same task via self-attention mechanism without using any internal memory could be highly non-trivial and potentially impossible. Languages such as $(aa)^*$ and Parity are among the simplest

[7] The results and limitations of single-layer Transformers are confined to this subsection. The rest of the results in the paper are not specific to single-layer Transformers unless explicitly mentioned.

[8] For tasks such as Parity, LSTMs can simply flip between two values in its hidden state upon receiving 1's as input and ignore when it receives 0's as input.

Language	Property	Transformer		LSTM	
		Bin 0	Bin 1	Bin 0	Bin 1
Parity	non-SF	68.7 (23.0)	0.0 (0.0)	100.0	100.0
$(aa)^*$	non-SF	100 (1.3)	0.0 (0.0)	100.0	100.0
$(abab)^*$	non-SF	100.0 (9.9)	5.4 (0.0)	100.0	100.0
$\mathcal{D}_1$	depth-1	100.0	100.0	100.0	100.0
$\mathcal{D}_2$	depth-2	74.6	3.1	100.0	100.0
$\mathcal{D}_4$	depth -4	90.2	3.3	100.0	100.0

Table 4: Results on non-star-free languages (non-SF) and the language $\mathcal{D}_n$. The values in parenthesis correspond to the scores obtained for a model without residual connections. This is to prevent the model from solving the task by memorizing the positional encodings and study the ability of self-attention mechanism to solve the task.

Encoding Scheme	$(aa)^*$		$(aaaa)^*$	
	Bin 0	Bin 1	Bin 0	Bin 1
Positional Masking	0.0	0.0	0.0	0.0
Absolute Encoding	1.3	0.0	6.7	0.0
Relative Encoding	0.6	0.0	1.7	0.0
$\cos(n\pi)$	100.0	100.0	0.0	0.0
Trainable Embedding	100.0	0.0	100.0	0.0

Table 5: Performance of transformer based models on $(aa)^*$ and $(aaaa)^*$, for different types of position encoding schemes. To separately study the effect of different position encodings on the self attention mechanism, we do not include residual connections in the models studied here.

non-star-free languages, and hence limitations in recognizing such languages carry over to a larger class of languages. The results above may suggest that the star-free languages are precisely the regular languages recognizable by Transformers. As we will see in the next section, this is not so.

7.1 Necessity of Positional Encodings

The architecture of Transformer imposes limitations for recognizing certain types of languages. Although Transformers seem to generalize well when they are capable of performing a task with only positional masking, they are incapable of recognizing certain types of languages without explicit positional encodings. We consider the family of star-free languages $\mathcal{D}_n$ defined in Sec. 3.1. Note that the task of recognizing $\mathcal{D}_n$ is equivalent to recognizing Dyck-1 with maximum depth n, where the symbols a and b in $\mathcal{D}_n$ are analogous to open and closing brackets in Dyck-1 respectively. The primary difference between recognizing $\mathcal{D}_n$ and Dyck-1 is that in case of $\mathcal{D}_n$, when the input reaches the maximum depth n, the model must predict a (the open bracket) as invalid for the next character, whereas in Dyck-1, open brackets are always allowed. We show that although Transformers with only positional masking can generalize well on Dyck-1, they are incapable of recognizing the language $\mathcal{D}_n$ for $n > 1$. The limitation arises from the fact that when the model receives a sequence of only a's, then due to the softmax based aggregation, the output of the self-attention block $\boldsymbol{a}_i$ will be a constant vector, implying that the output of the feed-forward will also be a constant vector, that is, $\boldsymbol{z}_1 = \boldsymbol{z}_2 = \ldots = \boldsymbol{z}_n$. In case of languages such as $\mathcal{D}_n$, if the input begins with n consecutive as, then, since the model cannot distinguish between the n-th a and the preceding a's, the model cannot recognize the language $\mathcal{D}_n$. This limitation does not exist if the model is provided explicit positional encoding. Upon evaluating Transformers with positional encodings on instances of the language $\mathcal{D}_n$, we found that the models are able to generalize to a certain extent on strings within the same lengths as seen during training but fail to generalize on higher lengths (Table 4). It is perhaps surprising that small and simpler self-attention networks can generalize very well on languages such as Dyck-1 but achieve limited performance on a language that belongs to a much simpler class such as star-free.

Similarly, since $(aa)^*$, is a unary language (alphabet size is 1), the model will always receive the same character at each step. Hence, for a model with only positional masking, the output vector will be the same at every step, making it incapable of recognizing the language $(aa)^*$. For the language Parity, when the input word contains only 1's, the task reduces to recognizing $(11)^*$ and hence a model without positional encodings is incapable of recognizing Parity even for very small lengths regardless of the size of the network (refer to Lemma C.4). We find it surprising that for Parity, which is permutation invariant, positional encodings are necessary for transformers to recognize them even for very small lengths.

7.2 Influence of Custom Positional Encodings

The capability and complexity of the network could significantly depend on the positional encoding scheme. For instance, for language $(aa)^*$, the ability of a self-attention network to recognize it depends solely on the positional encoding. Upon evaluating with standard absolute and relative encoding schemes, we observe that the model is unable to learn or generalize well. At the same time, it is easy to show that if $\cos(n\pi)$, which has a period of two

is used as positional encoding, the self-attention mechanism can easily achieve the task which we also observe when we empirically evaluated with such an encoding. However, the same encoding would not work for a language such as $(aaaa)^*$, which has a periodicity of four. Table 5 shows the performance of the model with different types of encodings. When we used fixed-length trainable positional embeddings, the obtained learned embeddings were very similar to the $\cos(n\pi)$ form; however, such embeddings cannot be used for sequences of higher lengths. This also raises the need for better learnable encodings schemes that can extrapolate to variable lengths of inputs not seen during training data such as (Liu et al., 2020).

Our experiments on over 15 regular languages seem to indicate that Transformers are able to generalize on star-free languages within dot-depth 1 but have difficulty with higher dot-depths or more complex classes like non-star-free languages. Table 9 in Appendix lists results on all considered regular languages.

8 Discussion

We showed that Transformers can easily generalize on certain counter languages such as Shuffle-Dyck and Boolean Expressions in a manner similar to our proposed construction. Our visualizations imply that Transformers do so with a generalizable mechanism instead of overfitting on some statistical regularities. Similar to natural languages, Boolean Expressions consist of recursively nested hierarchical constituents. Recently, Papadimitriou and Jurafsky (2020) showed that pretraining LSTMs on formal languages like Shuffle-Dyck transfers to LM performance on natural languages. At the same time, our results show clear limitations of Transformers compared to LSTMs on a large class of regular languages. Evidently, the performance and capabilities of Transformers heavily depend on architectural constituents e.g., the positional encoding schemes and the number of layers. Recurrent models have a more automata-like structure well-suited for counter and regular languages, whereas self-attention networks' structure is very different, which seems to limit their abilities for the considered tasks.

Our work poses a number of open questions. Our results are consistent with the hypothesis that Transformers generalize well for star-free languages with dot-depth 1, but not for higher depths. Clarifying this hypothesis theoretically and empirically is an attractive challenge. What does the disparity between the performance of Transformers on natural and formal languages indicate about the complexity of natural languages and their relation to linguistic analysis? (See also Hahn (2020)). Another interesting direction would be to understand whether certain modifications or recently proposed variants of Transformers improve their performance on formal languages. Regular and counter languages model some aspects of natural language while context-free languages model other aspects such as hierarchical dependencies. Although our results have some implications on them, we leave a detailed study on context-free languages for future work.

Acknowledgements

We thank the anonymous reviewers for their constructive comments and suggestions. We would also like to thank our colleagues at Microsoft Research and Michael Hahn for their valuable feedback and helpful discussions.

References

Satwik Bhattamishra, Arkil Patel, and Navin Goyal. 2020. On the computational power of transformers and its implications in sequence modeling. *arXiv preprint arXiv:2006.09286*.

Rina Cohen and Janusz Brzozowski. 1971. Dot-depth of star-free events. *Journal of Computer and System Sciences*, 5:1–16.

Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc Le, and Ruslan Salakhutdinov. 2019. Transformer-XL: Attentive language models beyond a fixed-length context. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 2978–2988, Florence, Italy. Association for Computational Linguistics.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. 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, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota. Association for Computational Linguistics.

Volker Diekert and Paul Gastin. 2008. First-order definable languages. In *Logic and Automata: History and Perspectives [in Honor of Wolfgang Thomas]*, volume 2 of *Texts in Logic and Games*, pages 261–306. Amsterdam University Press.

Patrick C Fischer, Albert R Meyer, and Arnold L Rosenberg. 1968. Counter machines and counter languages. *Mathematical systems theory*, 2(3):265–283.

Felix A Gers and E Schmidhuber. 2001. Lstm recurrent networks learn simple context-free and context-sensitive languages. *IEEE Transactions on Neural Networks*, 12(6):1333–1340.

Michael Hahn. 2020. Theoretical limitations of self-attention in neural sequence models. *Transactions of the Association for Computational Linguistics*, 8:156–171.

Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation*, 9(8):1735–1780.

Gerhard Jäger and James Rogers. 2012. Formal language theory: refining the chomsky hierarchy. *Philosophical Transactions of the Royal Society B: Biological Sciences*, 367(1598):1956–1970.

John F Kolen and Stefan C Kremer. 2001. *A field guide to dynamical recurrent networks*. John Wiley & Sons.

Samuel A Korsky and Robert C Berwick. 2019. On the computational power of rnns. *arXiv preprint arXiv:1906.06349*.

Xuanqing Liu, Hsiang-Fu Yu, Inderjit Dhillon, and Cho-Jui Hsieh. 2020. Learning to encode position for transformer with continuous dynamical model. *arXiv preprint arXiv:2003.09229*.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*.

Robert McNaughton and Seymour A. Papert. 1971. *Counter-Free Automata (M.I.T. Research Monograph No. 65)*. The MIT Press.

William Merrill. 2019. Sequential neural networks as automata. In *Proceedings of the Workshop on Deep Learning and Formal Languages: Building Bridges*, pages 1–13, Florence. Association for Computational Linguistics.

William Merrill. 2020. On the linguistic capacity of real-time counter automata. *arXiv preprint arXiv:2004.06866*.

William Merrill, Gail Weiss, Yoav Goldberg, Roy Schwartz, Noah A Smith, and Eran Yahav. 2020. A formal hierarchy of rnn architectures. *arXiv preprint arXiv:2004.08500*.

Joshua J. Michalenko, Ameesh Shah, Abhinav Verma, Swarat Chaudhuri, and Ankit B. Patel. 2019. Finite automata can be linearly decoded from language-recognizing RNNs. In *International Conference on Learning Representations*.

Isabel Papadimitriou and Dan Jurafsky. 2020. Pretraining on non-linguistic structure as a tool for analyzing learning bias in language models. *arXiv preprint arXiv:2004.14601*.

Jorge Pérez, Javier Marinković, and Pablo Barceló. 2019. On the turing completeness of modern neural network architectures. In *International Conference on Learning Representations*.

Jean-ric Pin. 2017. The dot-depth hierarchy, 45 years later. In *The Role of Theory in Computer Science*, pages 177–201.

Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. 2018. Improving language understanding by generative pre-training. *URL https://s3-us-west-2. amazonaws. com/openai-assets/researchcovers/languageunsupervised/language understanding paper. pdf*.

Emily Reif, Ann Yuan, Martin Wattenberg, Fernanda B Viegas, Andy Coenen, Adam Pearce, and Been Kim. 2019. Visualizing and measuring the geometry of bert. In *Advances in Neural Information Processing Systems*, pages 8592–8600.

Anna Rogers, Olga Kovaleva, and Anna Rumshisky. 2020. A primer in bertology: What we know about how bert works. *arXiv preprint arXiv:2002.12327*.

Luzi Sennhauser and Robert Berwick. 2018. Evaluating the ability of LSTMs to learn context-free grammars. In *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, pages 115–124, Brussels, Belgium. Association for Computational Linguistics.

Tao Shen, Tianyi Zhou, Guodong Long, Jing Jiang, Shirui Pan, and C. Zhang. 2018. Disan: Directional self-attention network for rnn/cnn-free language understanding. In *AAAI*.

Natalia Skachkova, Thomas Trost, and Dietrich Klakow. 2018. Closing brackets with recurrent neural networks. In *Proceedings of the 2018 EMNLP Workshop BlackboxNLP: Analyzing and Interpreting Neural Networks for NLP*, pages 232–239, Brussels, Belgium. Association for Computational Linguistics.

Howard Straubing. 1994. *Finite Automata, Formal Logic, and Circuit Complexity*. Birkhauser Verlag, CHE.

Mirac Suzgun, Yonatan Belinkov, Stuart Shieber, and Sebastian Gehrmann. 2019a. LSTM networks can perform dynamic counting. In *Proceedings of the Workshop on Deep Learning and Formal Languages: Building Bridges*, pages 44–54, Florence. Association for Computational Linguistics.

Mirac Suzgun, Yonatan Belinkov, and Stuart M. Shieber. 2019b. On evaluating the generalization of LSTM models in formal languages. In *Proceedings of the Society for Computation in Linguistics (SCiL) 2019*, pages 277–286.

Yao-Hung Hubert Tsai, Shaojie Bai, Makoto Yamada, Louis-Philippe Morency, and Ruslan Salakhutdinov. 2019. Transformer dissection: An unified understanding for transformer's attention via the lens of kernel. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 4344–4353, Hong Kong, China. Association for Computational Linguistics.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008.

Elena Voita, David Talbot, Fedor Moiseev, Rico Sennrich, and Ivan Titov. 2019. Analyzing multi-head self-attention: Specialized heads do the heavy lifting, the rest can be pruned. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 5797–5808, Florence, Italy. Association for Computational Linguistics.

Qinglong Wang, Kaixuan Zhang, Alexander G. Ororbia II, Xinyu Xing, Xue Liu, and C. Lee Giles. 2018a. A comparative study of rule extraction for recurrent neural networks. *CoRR*, abs/1801.05420v2.

Qinglong Wang, Kaixuan Zhang, II Ororbia, G Alexander, Xinyu Xing, Xue Liu, and C Lee Giles. 2018b. A comparative study of rule extraction for recurrent neural networks. *arXiv preprint arXiv:1801.05420*.

Alex Warstadt, Yu Cao, Ioana Grosu, Wei Peng, Hagen Blix, Yining Nie, Anna Alsop, Shikha Bordia, Haokun Liu, Alicia Parrish, Sheng-Fu Wang, Jason Phang, Anhad Mohananey, Phu Mon Htut, Paloma Jeretic, and Samuel R. Bowman. 2019. Investigating BERT's knowledge of language: Five analysis methods with NPIs. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 2877–2887, Hong Kong, China. Association for Computational Linguistics.

Gail Weiss, Yoav Goldberg, and Eran Yahav. 2018. 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 (Volume 2: Short Papers)*, pages 740–745, Melbourne, Australia. Association for Computational Linguistics.

Gail Weiss, Yoav Goldberg, and Eran Yahav. 2019. Learning deterministic weighted automata with queries and counterexamples. In H. Wallach, H. Larochelle, A. Beygelzimer, F. AlcheBuc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8560–8571. Curran Associates, Inc.

Baosong Yang, Longyue Wang, Derek F. Wong, Lidia S. Chao, and Zhaopeng Tu. 2019. Assessing the ability of self-attention networks to learn word order. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 3635–3644, Florence, Italy. Association for Computational Linguistics.

Chulhee Yun, Srinadh Bhojanapalli, Ankit Singh Rawat, Sashank Reddi, and Sanjiv Kumar. 2020. Are transformers universal approximators of sequence-to-sequence functions? In *International Conference on Learning Representations*.

A Roadmap

The appendix is organized as follows. In section B we first provide formal definitions of the key languages used in our investigation in the main paper. In sections B.1 and B.2, we also provide the formal definitions of automata, star-free languages and the dot-depth hierarchy. In section C, we provide the details of all our expressiveness results. Section D contains the details of our experimental setup which could be relevant for reproducibility of the results and includes a thorough discussion of the choice of character prediction task. The list of all the formal languages we have considered, their dataset statistics as well as the results are provided in section D.

B Definitions

In this section, we provide formal definitions of some of the languages used in our analysis. In counter languages, we first define the family of shuffled Dyck-1 languages. The language Dyck-1 is a simple context-free language that can also be recognized by a counter automaton with a single counter. We generate the data for Dyck-1 based on the following PCFG,

$$S \rightarrow \begin{cases} (S) & \text{with probability } p \\ SS & \text{with probability } q \\ \varepsilon & \text{with probability } 1-(p+q) \end{cases}$$

where $0 < p, q < 1$ and $(p+q) < 1$. We use 0.5 as the value of p and 0.25 as the value for q.

Shuffle-Dyck. We now define the Shuffle-Dyck language introduced and described in (Suzgun et al., 2019a). We first define the shuffling operation formally. The shuffling operation $||$: $\Sigma^* \times \Sigma^* \rightarrow \mathcal{P}(\Sigma^*)$ can be inductively defined as follows:[9]

- $u||\varepsilon = \varepsilon||u = \{u\}$
- $\alpha u||\beta v = \alpha(u||\beta v) \cup \beta(\alpha u||v)$

for any $\alpha, \beta \in \Sigma$ and $u, v \in \Sigma^*$. For instance, the shuffle of ab and cd is

$$ab||cd = \{abcd, acbd, acdb, cabd, cadb, cdab\}.$$

There is a natural extension of the shuffling operation $||$ to languages. The *shuffle* of two languages $\mathcal{L}_1$ and $\mathcal{L}_2$, denoted $\mathcal{L}_1||\mathcal{L}_2$, is the set of all possible interleavings of the elements of $\mathcal{L}_1$ and $\mathcal{L}_2$, respectively, that is:

$$\mathcal{L}_1||\mathcal{L}_2 = \bigcup_{u \in \mathcal{L}_1,\ v \in \mathcal{L}_2} u||v$$

Given a language $\mathcal{L}$, we define its self-shuffling $\mathcal{L}||^2$ to be $\mathcal{L}||\sigma(\mathcal{L})$, where σ is an isomorphism on the vocabulary of $\mathcal{L}$ to a disjoint vocabulary. More generally, we define the k-self-shuffle

$$\mathcal{L}||^k = \begin{cases} \{\varepsilon\} & \text{if } k = 0 \\ \mathcal{L}||\sigma(\mathcal{L}||^{k-1}) & \text{otherwise} \end{cases} .$$

We use Shuffle-k to denote the shuffle of k Dyck-1 languages (Dyck-$1||^k$) each with its own brackets. Shuffle-1 is the same as Dyck-1. For instance the language Shuffle-2 is the shuffle of Dyck-1 over alphabet $\Sigma = \{(,)\}$ and another Dyck-1 over the alphabet $\Sigma = \{[,]\}$. Hence the resulting Shuffle-2 language is defined over alphabet $\Sigma = \{[,], (,)\}$ and contains words such as $([])$ and $[(())]$ but not $])[($. This is different from the context-free language Dyck-2 in which $([])$ belongs to the language but $([)]$ does not. Similar to (Suzgun et al., 2019a) we generate the training data by generating sequence for Dyck-n but by providing the correct target values for the character prediction task.

n-ary Boolean Expressions. We now define the family of languages n-ary Boolean Expressions parameterized by the number and arities of its operators. An instance of the language contains operators of different arities and as shown in (Fischer et al., 1968), these languages can be recognized by counter-machines with a single counter. However as opposed to Dyck-1 the values with which the counters will be incremented or decremented will depend on the arity of its operator. A language with n operators can be defined by the following derivation rules

```
<exp> -> <VALUE>
<exp> -> <UNARY> <exp>
<exp> -> <BINARY> <exp> <exp>
..
<exp> -> <n-ARY> <exp> .. <exp>
```

Tomita Grammars Tomita Grammars are 7 regular langauges defined on the alphabet $\Sigma = \{0, 1\}$. Tomita-1 has the regular expression 1^* i.e. the strings containing only 1's and no 0s are allowed. Tomita-2 is defined by the regular expression $(10)^*$. Tomita-3 accepts the strings where odd number

[9] We abuse notation by allowing a string to stand for the singleton containing that string. ϵ is the empty string.

of consecutive 1s are always followed by an even number of 0s. Tomita-4 accepts the strings that do not contain 3 consecutive 0s. In Tomita-5 only the strings containing an even number of 0s and even number of 1s are allowed. In Tomita-6 the difference in the number of 1s and 0s should be divisible by 3 and finally, Tomita-7 has the regular expression $0^*1^*0^*1^*$.

We note that Tomita $2 = \mathcal{D}_1 = (01)^*$ and that the very simple language $\{0, 1, 2\}^*02^*$ has dot-depth 2 (Cohen and Brzozowski, 1971).

B.1 Counter Automata

We define the general counter machine following (Fischer et al., 1968). We are concerned with real-time counter machines here in which the number of computation steps is bounded by the number of inputs similar to how we use sequence models in practice. The machine has a finite number of unbounded counters and it modifies it by adding or subtracting values or resetting the counter value to 0. For $m \in \mathbb{Z}$, let $+m$ denote the function $x \mapsto x + m$. Let $\times 0$ denote the constant zero function $x \mapsto 0$.

Definition B.1 (General counter machine (Fischer et al., 1968))**.** A k-counter machine is a tuple $\langle \Sigma, Q, q_0, u, \delta, F \rangle$ with

1. A finite alphabet Σ
2. A finite set of states Q
3. An initial state q_0
4. A counter update function
$$u : \Sigma \times Q \times \{0, 1\}^k \to \left(\{+m : m \in \mathbb{Z}\} \cup \{\times 0\}\right)^k$$
5. A state transition function
$$\delta : \Sigma \times Q \times \{0, 1\}^k \to Q$$
6. An acceptance mask
$$F \subseteq Q \times \{0, 1\}^k$$

A machine processes an input string x one token at a time. For each token, we use u to update the counters and δ to update the state according to the current input token, the current state, and a finite mask of the current counter values.

For a vector $\boldsymbol{v}$, let $z(\boldsymbol{v})$ denote the broadcasted "zero-check" function, i.e. $z(\boldsymbol{v})_i$ is 0 if $v_i = 0$ or 1 otherwise. Let $\langle q, \boldsymbol{c} \rangle \in Q \times \mathbb{Z}^k$ be a configuration of machine M. Upon reading input $x_t \in \Sigma$, we define the transition

$$\langle q, \boldsymbol{c} \rangle \to_{x_t} \langle \delta(x_t, q, z(\boldsymbol{c})), u(x_t, q, z(\boldsymbol{c}))(\boldsymbol{c}) \rangle.$$

For any string $x \in \Sigma^*$ with length n, a counter machine accepts x if there exist states $q_1, .., q_n$ and counter configurations $\boldsymbol{c}_1, .., \boldsymbol{c}_n$ such that

$$\langle q_0, \mathbf{0} \rangle \to_{x_1} \langle q_1, \boldsymbol{c}_1 \rangle \to_{x_2} .. \to_{x_n} \langle q_n, \boldsymbol{c}_n \rangle \in F.$$

A counter machines accepts a language L if, for each $x \in \Sigma^*$, it accepts x iff $x \in L$. Refer to (Merrill, 2020) for more details on counter machines, variants and their properties.

B.2 Star-free regular languages and the dot-depth hierarchy

Star-free regular languages (defined in the main paper) are a simpler subclass of regular languages; they have regular expressions without Kleene star (but use set complementation). The set of star-free languages is further stratified by the dot-depth hierarchy, which is a hierarchy of families of languages whose union is the family of star-free languages. Informally, the position of a language in this hierarchy is a measure of the number of nested concatenations or sequentiality required to express the language in a star-free regular expression. Both the star-free regular languages as well as the dot-depth hierarchy are well-studied with rich connections and multiple (equivalent) definitions. For more information, see e.g. (McNaughton and Papert, 1971; Cohen and Brzozowski, 1971; Straubing, 1994; Diekert and Gastin, 2008; Jäger and Rogers, 2012; Pin, 2017).

To define the dot-depth hierarchy, we first define Boolean and concatenation closures of language families. For a language family $\mathcal{L}$ over a finite alphabet $\Sigma = \{a_1, \ldots, a_k\}$, its Boolean closure $\mathcal{BL}$ is the set of languages obtained by applying Boolean operators (union, intersection and set complementation w.r.t. Σ^*) to the languages in $\mathcal{L}$. In other words, $\mathcal{BL}$ is the smallest family of languages containing $\mathcal{L}$ and closed under Boolean operations: if $L_1, L_2 \in \mathcal{L}$ then $L_1 \cap L_2 \in \mathcal{BL}$ and $L_1 \cup L_2 \in \mathcal{BL}$ and $L_1^c, L_2^c \in \mathcal{BL}$. Similarly, define the concatenation closure of $\mathcal{L}$ as the smallest family of languages containing $\mathcal{L}$ and closed under concatenation: if $L_1, L_2 \in \mathcal{L}$ then $L_1 L_2 \in \mathcal{ML}$.

We begin with the class $\mathcal{E}$ of basic languages consisting of $\{a_1\}, \ldots \{a_k\}, \{\epsilon\}, \emptyset$. By alternately applying the operators $\mathcal{B}$ and $\mathcal{M}$ to $\mathcal{E}$ we can define the hierarchy

$$\mathcal{E} \subseteq \mathcal{ME} \subseteq \mathcal{BME} \subseteq \mathcal{MBME} \subseteq \ldots.$$

Let $\mathcal{B}_0 = \mathcal{BME}$. The dot-depth hierarchy is the sequence of families of languages $\mathcal{B}_0 \subseteq \mathcal{B}_1 \subseteq \ldots$ defined inductively by $\mathcal{B}_{n+1} = \mathcal{MB}_n$ for $n \geq 0$. It is known that all the inclusions in $\mathcal{B}_0 \subseteq \mathcal{B}_1 \subseteq \ldots$ are strict and is exemplified by the languages $\mathcal{D}_n$ (see Pin (2017)). Minor variations in the definition exist in the literature; in particular, we could have applied the operator $\mathcal{B}$ first, but these have only minor effects on the overall concept and results.

C Expressiveness Results

We define a weaker version of counter automata which are restricted in a certain sense. Then, we show that Transformers are at least as powerful as such automata.

Definition C.1 (Simplified and Stateless counter machine)**.** We define a counter machine to be simplified and stateless if u and δ have the following form,

$$u : \Sigma \to \{+m : m \in \mathbb{Z}\}^k,$$

$$\delta : \Sigma \to Q$$

This implies that the machine can have k counters. The counters can be incremented or decremented by any values but it will only depend on the input symbol. Similarly, the state transition will also depend on the current input. A string $x \in \Sigma^*$ will be accepted if $\langle q_n, z(\boldsymbol{c}_n)\rangle \in F$. We use L_{RCL} to denote the class of languages recognized by such a counter machine. The above language is similar to Σ-restricted counter machine defined in (Merrill et al., 2020).

Lemma C.1. *Transformers can recognize L_{RCL}.*

Proof. Let $s_1, s_2, \ldots, s_n$ denote a sequence $w \in \Sigma^*$. If the counter machine has k counters, then let the dimension of intermediate vectors $d_{model} = 2k + |\Sigma|$. The first $2k$ dimensions will be reserved for counter related operations and then $|Q|$ dimensions will be reserved to obtain the state vector. The embedding vector $\boldsymbol{x}_i$ of each symbol will have 0s in the first $2k$ dimensions and the last $|\Sigma|$ dimensions will have the one-hot encoding representation of the symbol. For a k counter machine the value vectors would have a subvector of dimension 2 reserved for computations pertaining to each of the counter. That is, $\boldsymbol{x}_{2j:2j+1}$ will be reserved for the jth counter where $0 \leq j < k$. For any given input symbol s, if $u(s)$ has counter operation of $+m$ at the jth counter, then the value will be such that $\boldsymbol{v}$ will contain $+m$ at index $2j$ and $-m$ at index $2j+1$ upto index $2k$. The last $|\Sigma|$ dimensions will have the value 0 in the value vectors. This can be easily obtained by a linear transformation $V(.)$ over one-hot encodings. The linear transformation $K(.)$ to obtain the key vectors will lead to zero vectors and hence all inputs will have equal attention weights. The linear transformation $V(.)$ to obtain the value vectors $\boldsymbol{v}_i$ will be identity function. Hence the output of the self-attention block along with residual connection will be of the form $\boldsymbol{a}_i = \frac{1}{i}\sum_{t=1}^{i} \boldsymbol{v}_t + \boldsymbol{x}_i$.

The last $|\Sigma|$ dimensions of the vector $\boldsymbol{a}_i$ will have one-hot encoding of the input vector at i-th step. The one-hot encoding of the input can be easily mapped to the one-hot encoding for the corresponding state using a simple FFN. Additionally, this will ensure that, at the i-th step, the output of the self-attention block $\boldsymbol{a}_i$ will have the value $\frac{c_j}{i}$ at indices $2j$, where c_j denotes the counter value of the counter automata representing the language. Similarly, the odd indices $2j+1$ will have the value $-\frac{c_j}{i}$. After applying a simple feed-forward network with ReLU activation, we obtain the output vector $\boldsymbol{z}_i$. It is easy to implement the zero check function with a simple linear layer over the output vector. The network accepts an input sequence w when the values in the output vector corresponding to each counter and state at the n-th correspond to that required for the final state.

□

We next show that n-ary Boolean Expressions can be recognized by Transformers with a similar construction.

Lemma C.2. *Transformers can recognize n-ary Boolean Expressions.*

Proof. Let L_m denote a language of type n-ary Boolean Expressions with m operators defined over the alphabet Σ. Consider a single layer Transformer network with $d_{model} = 2$. Let $s_0, s_1, \ldots, s_n$ be sequence w where $w \in \Sigma^*$. Let s_0 be a special start symbol with embedding $f_e = [+1, -1]$. The embeddings of each input symbol $s \in \Sigma$ are defined as follows, $f_e(s) =$

$[+(r-1), -(r-1)]$ where r denotes the arity of the symbol. The arity of values such as 0 and 1 is taken as 0. Similar to the previous construction, the key values are null and hence attention weights are uniform leading to $\boldsymbol{a}_i = \frac{1}{i}\sum_{t=1}^{i} \boldsymbol{v}_t$. Hence the output of the self-attention block will be $\boldsymbol{a}_i = [\frac{c_j}{i}, -\frac{c_j}{i}]$, where c_j denotes the counter value of the automata representing the language. Essentially, for each operator, the value added to the counter is equal to its arity subtracted by 1. For each value such as 0 and 1, the counter value is decremented by 1. We then apply a simple FFN with ReLU activation to obtain the output vector $\boldsymbol{z}_i = \mathsf{ReLU}(\mathbf{I}\boldsymbol{a}_i)$.

An input sequence w belongs to the language L_m if the second coordinate of the output is zero at every step, that is, $\boldsymbol{z}_{i,2} = 0$ for $0 \leq i \leq n$ and $\boldsymbol{z}_n = \mathbf{0}$. □

Let Reset-Dyck-1 be a language defined over alphabet $\Sigma = \{[,], 1\}$, where 1 denotes a symbol that requires a reset operation. Words in Reset-Dyck-1 have the form $\Sigma^* 1 v$, where the string v belongs to Dyck-1. So essentially, when the machine encounters the reset symbol 1, it has to ignore all the previous inputs, reset the counter to 0 and go to start state.

Lemma C.3. *A single-layer Transformer with only positional masking cannot recognize the language Reset-Dyck-1.*

Proof. The proof is straightforward. Let $s_1, s_2, \ldots, s_n$ be an input sequence w. Let s_r denote the r-th symbol where the reset symbol occurs. It is easy to see that the scoring function $\langle \boldsymbol{q}_n, K(\boldsymbol{v}_r)\rangle$ is independent of the position as well as the inputs before the reset symbol which are relevant for the reset operation. Consider the case where the first half of the input contains a sequence of open and closing brackets such that it does not belong to Dyck-1 and the second half contains a sequence that belongs to Dyck-1. If the reset symbol occurs after the first half of the sequence, then the word belongs to Dyck-1 and if it occurs in the beginning then it does not belong to the language Dyck-1. However, by construction, the output of the model $\boldsymbol{z}_n$ will remain the same regardless of the position of the reset symbol and hence by contradiction, it cannot recognize such a language. □

The above limitation does not exist if there is a two layer network. The scoring function as well as value vector of the reset symbol will be dependent of the inputs that precede it. Hence it is not necessary that a two layer network will not be able to recognize such a language. Indeed, as shown in the main paper, the 2-layer Transformer performs well on Reset-Dyck-1.

Lemma C.4. *Transformers with only positional masking cannot recognize the language* $(aa)^*$.

Proof. Let $s_1, s_2, \ldots, s_n$ be an input sequence w where $w \in a^*$. Since it is a unary language, the input at each step will be the same symbol and hence the embedding as well as query, key and value vectors will be the same. Since all the value vectors are the same, regardless of the attention weights, the output of the self-attention vector $\boldsymbol{a}_i$ will be a constant vector at each timestep. This implies that the output vectors $\boldsymbol{z}_1 = \boldsymbol{z}_2 = \ldots = \boldsymbol{z}_n$. Inductively, it is easy to see that regardless of the number of layers this phenomenon will carry forward and hence the output vector at each timestep will be the same. Thus, the network cannot distinguish output at even steps and odd steps which is necessary to recognize the language $(aa)^*$.

□

For parity, in the case where the input consists of only 1s, the problem reduces to recognizing $(11)^*$. Hence it follows from the above result that a network without positional encoding cannot recognize parity even for minimal lengths.

D Experiments

D.1 Discussion on Character Prediction Task

As described in section 5.1, we use character prediction task in our experiments to evaluate the model's ability to recognize a language. In character prediction task the model is only presented with positive samples from a given language and its goal is to predict the next set of valid characters. During inference, the model predicts the next set of legal characters at each step and a prediction is considered to be correct if and only if the model's output at every step is correct. The character prediction task is similar to predicting which of the input characters are allowed to make a transition in a given automaton such that it leads to a non-dead state. If an input character is not among the legal characters, that implies the underlying automaton will transition to a dead state and regardless of the following characters, the input word will never be accepted. When the end-of-sequence symbol is allowed as

one of the next set of legal characters, it implies that the underlying automaton is in the final state and the input can be accepted.

Character prediction and classification. If a model can perform character prediction task perfectly, then it can also perform classification in the following way. For an input sequence $s_1, s_2, \ldots, s_n$, the model receives the sequence $s_1, \ldots, s_i$ for $1 \leq i \leq n$ at each step i and model predicts the set of valid characters in the $(i+1)^{th}$ position. If the next character is among the model's predicted set of valid characters at each step i and the end of symbol character is allowed at the n-th step, then the word is accepted and if any character is not within the model's predicted set of valid characters, then the word is rejected. One of the primary reason for the choice of character prediction task is that it is arguably more robust than the standard classification task. The metric for character prediction task is relatively stringent and the model is required to model the underlying mechanism as opposed to just one label in standard classification. Note that the null accuracy (accuracy when all the predictions are replaced by a single label) is 50% if the distribution of labels is balanced (higher otherwise), on the other hand the null accuracy of character prediction task is close to 0. Additionally, in case of classification, depending on how the positive or negative data are generated, the model may also be biased to predict based on some statistical regularites instead of modeling the actual mechanism. In (Weiss et al., 2019), they find that LSTMs trained to recognize Dyck-1 via classification on randomly sampled data do not learn the correct mechanism and fail on adversarially generated samples. On the other hand, Suzgun et al. (2019a) show that LSTMs trained to recognize Dyck-1 via character prediction task learn to perform the correct mechanism required to do the task.

Character prediction and language modelling. The character prediction task has clear connections with Language modelling. If a model can perform language modelling perfectly, then it can perform character prediction task in the following way. For an input sequence $s_1, s_2, \ldots, s_n$, the model receives the sequence $s_1, \ldots, s_i$, for $1 \leq i \leq n$ at each step i and predicts a distribution over the vocabulary. Mapping all the characters for which the model assigns a nonzero probability to 1 and mapping to 0 for all characters that are assigned zero probability will reduce it to character prediction task. However, there are a few issues with using language modelling in our formal language setting. Firstly, as mentioned in (Suzgun et al., 2019a), the task of recognizing a language is not inherently probabilistic. Our goal here is to understand whether a network can or cannot model a particular language. Using language modelling will require us to impose a distribution arbitrarily for the given setting. More importantly, in character prediction task, some signals are explicitly provided. In the case of language modelling, we may just have to rely on the model to pick up those nuanced signals. For instance, in the language $\mathcal{D}_n$, when the input reaches the maximum depth n, in character prediction task it is explicitly provided the target value that a is not allowed anymore whereas in language modelling the model is expected to assign zero probability to a at the maximum depth based on the fact that it will never see a word depth more than n in the training data. This phenomenon has major issues. For instance, when we consider Dyck-1 in practical setting, we can only provide it with limited data which implies there will be a sequence with a maximum finite depth. In this scenario, a language model trained on such data may learn the Dyck-1 language or the language $\mathcal{D}_n$ with that particular maximum depth. This limitation does not exist in the character prediction task where the signal is explicitly provided during training.

D.2 Experimental Details

We use 4 NVIDIA Tesla P100 GPUs each with 16 GB memory to run our experiments, and train and evaluate our models on about 9 counter languages and 18 regular languages. The important details of all of these languages like the training and test sizes and the lengths of the strings considered, have been summarized in Table 6. In all of our experiments, the first bin always has the same length range as the training set, i.e. if the training set contains strings with lengths in range $[2, 50]$, then the strings in the first test bin will also lie in the same range. Width of bin is the difference between upper and lower limits of the string lengths that lie in that bin. All the test bins are taken to be disjoint from each other. Hence, if we have 3 bins with a width of 50 and the training range is $[2, 50]$, then the length ranges for the test bins will be $[2, 50]$, $[52, 100]$ and $[102, 150]$.

Language	Training Data		Test Data			
	Size	Length Range	Size per Bin	Length Range	Number of Bins	Bin Width
Counter Languages						
Shuffle-2	10000	[2, 50]	2000	[2, 150]	3	50
Shuffle-4	10000	[2, 100]	2000	[2, 200]	3	50
Shuffle-6	10000	[2, 1000]	2000	[2, 200]	3	50
Boolean-3	10000	[2, 50]	2000	[2, 150]	3	50
Boolean-5	10000	[2, 50]	2000	[2, 150]	3	50
$a^n b^n$	50	[2, 100]	50	[2, 300]	3	100
$a^n b^n c^n$	50	[3, 150]	50	[3, 450]	3	150
$a^n b^n c^n d^n$	50	[4, 200]	50	[4, 600]	3	200
Dyck-1	10000	[2, 50]	2000	[2, 150]	3	50
Regular Languages						
Tomita 1	50	[2, 50]	100	[2, 100]	2	50
Tomita 4	10000	[2, 50]	2000	[2, 100]	2	50
Tomita 7	10000	[2, 50]	2000	[2, 100]	2	50
Tomita 2	25	[2, 50]	50	[2, 100]	2	50
$aa^*bb^*cc^*dd^*ee^*$	10000	[5, 200]	1000	[5, 300]	2	100
$\{a, b\}^* d \{b, c\}^*$	10000	[1, 50]	2000	[1, 100]	2	50
$\{0, 1, 2\}^* 02^*$	10000	[2, 50]	2000	[2, 100]	2	50
$\mathcal{D}_2$	10000	[2, 100]	2000	[2, 200]	2	100
$\mathcal{D}_3$	10000	[2, 100]	2000	[2, 200]	2	100
$\mathcal{D}_4$	10000	[2, 100]	2000	[2, 200]	2	100
$\mathcal{D}_{12}$	10000	[2, 100]	2000	[2, 200]	2	100
Parity	10000	[2,50]	2000	[2, 100]	2	50
$(aa)^*$	250	[2, 500]	50	[2, 600]	2	100
$(aaaa)^*$	125	[4, 500]	25	[4, 600]	2	100
$(abab)^*$	125	[4, 500]	25	[4, 600]	2	100
Tomita 3	10000	[2, 50]	2000	[2, 100]	2	50
Tomita 5	10000	[2, 50]	2000	[2, 100]	2	50
Tomita 6	10000	[2, 50]	2000	[2, 100]	2	50

Table 6: Statistics of different datasets used in the experiments. Note that the width of the first bin is always defined by the training set (see D), and hence can be different from the widths of other bins reported in Bin Width column. As an example, for $(aa)^*$, the first bin will have a length range of [2, 500] and [502, 600] for the second bin.

For each of these languages, we extensively tune on a bunch of different architectural and optimization related hyperparameters. Table 7 lists the hyperparameters considered in our experiments and the bounds for each of them. This corresponds to about 162 different configurations for tuning transformers (for a hidden size of 3, 4 heads are not allowed) and 40 configurations for LSTMs . Over all the languages and hyperparameters there were a minimum of 117 parameters and a maximum of 17,888 parameters for the models that we considered. We use a grid search procedure to tune the hyperparameters. While reporting the accuracy scores for a given language, we compute the mean of the top 5 accuracies, corresponding to all hyperparameter configurations. For some experiments we had to consider the hyperparameters lying outside of the values specified in Table 7. As an instance, we considered 4 layer transformers in the cases where the training accuracies obtained were low for single and two layered networks and reported the results accordingly.

For training our models we used RMSProp optimizer with the smoothing constant $\alpha = 0.99$. In our initial few experiments we also tried Stochastic Gradient Descent with learning rate decay and Adam Optimizer, but decided to go ahead with RMSProp as it outperformed SGD in majority of experiments and gave similar performance as Adam but needed fewer hyperparameters. For each language we train models corresponding to each language for 100 epochs and a batch size of 32. In case of convergence, i.e. perfect accuracies for all the bins, before completion of all epochs, we stop the training process early. The results of our experiments on counter and regular languages are provided in Tables 8 and 9 respectively.

Hyperparameter	Bounds
Hidden Size	[3, 32]
Heads	[1, 4]
Number of Layers	[1, 2] — [1, 4]
Learning Rate	[1e-2, 1e-3]
Position Encoding Scheme	[Absolute, Relative, Positional Masking]

Table 7: Different hyperparameters and the values considered for each of them. Note that certain parameters like Heads and Position Encoding Scheme are only relevant for Transformer based models and not for LSTMs. We considered upto 4 layers transformers in the cases where the training accuracies obtained were low for single and two layered networks and reported the results accordingly.

E Plots

We visualize different aspects of the trained models to understand how they achieve a particular task and if the learned behaviour resembles our constructions. Figure 3 shows the value vectors corresponding to the trained models on Shuffle-2 and Boolean-3 Language. We also visualize the attention weights corresponding to these two models in Figure 4. Similar to the self-attention output visualizations for Shuffle-2 and Boolean-3 in the main paper, we visualize these values for a model trained on Shuffle-4 in Figure 5 and again, find close correlations with the depth to length ratios of different types of brackets in the language. Finally, in Figure 6, we visualize a component of the learned position embeddings vectors and found a similar behaviour to $\cos(n\pi)$ agreeing with our hypothesis.

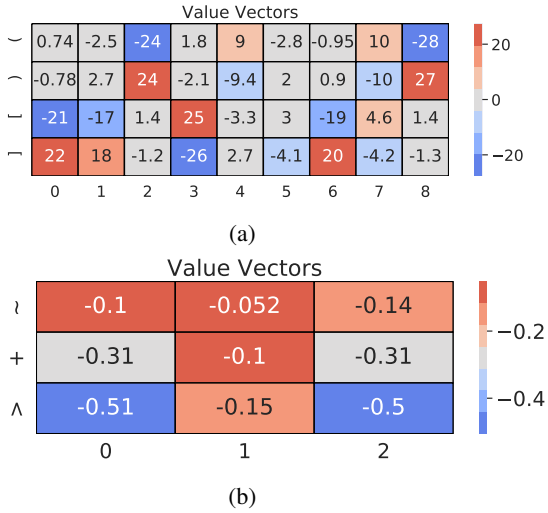
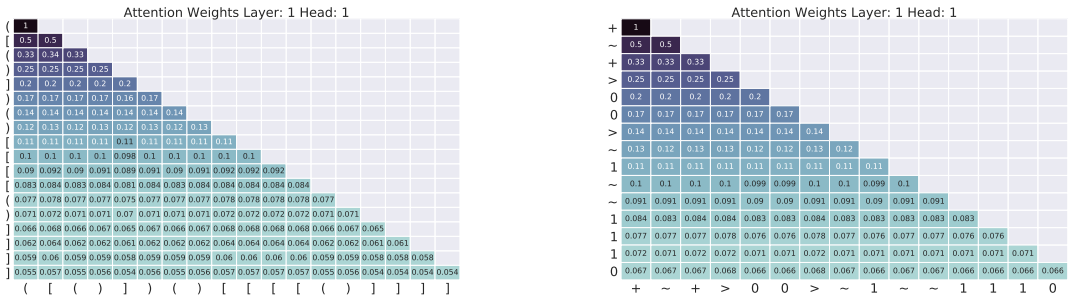


Figure 3: Plot of value vectors of transformer based models trained on Shuffle-2 3a and Boolean-3 language 3b. The Shuffle-2 model had a hidden size of 8 and boolean-3 model had a hidden size of 3. The x-axis corresponds to different components of the value vectors for both models. Shuffle-2 language consisted of square and round brackets, while for Boolean-3 we considered 3 operators namely: $\sim$ a unary operator, $+$ a binary operator and finally, $>$ which is a ternary operator..

Language	Model	Bin-1 Accuracy [1, 50]↑	Bin-2 Accuracy [51, 100]↑	Bin-3 Accuracy [101, 150]↑
Dyck-1	**LSTM (Baseline)**	**100.0**	**100.0**	**100.0**
	Transformer (Absolute Positional Encodings)	**100.0**	**100.0**	**100.0**
	Transformer (Relative Positional Encodings)	**100.0**	91.0	60.7
	Transformer (Only Positional Masking)	**100.0**	**100.0**	**100.0**
Shuffle-2	**LSTM (Baseline)**	**100.0**	**100.0**	**100.0**
	Transformer (Absolute Positional Encodings)	**100.0**	85.2	63.3
	Transformer (Relative Positional Encodings)	**100.0**	51.6	3.8
	Transformer (Only Positional Masking)	**100.0**	**100.0**	93.0
Shuffle-4	**LSTM (Baseline)**	**100.0**	**100.0**	**99.6**
	Transformer (Absolute Positional Encodings)	**100.0**	46.6	20.8
	Transformer (Relative Positional Encodings)	**100.0**	57.2	5.5
	Transformer (Only Positional Masking)	**100.0**	**100.0**	98.8
Shuffle-6	**LSTM (Baseline)**	**100.0**	**99.9**	99.5
	Transformer (Absolute Positional Encodings)	**100.0**	50.4	16.6
	Transformer (Relative Positional Encodings)	**100.0**	59.1	5.7
	Transformer (Only Positional Masking)	**100.0**	**99.9**	94.0
Boolean Expressions (3)	**LSTM (Baseline)**	**100.0**	**100.0**	99.7
	Transformer (Absolute Positional Encodings)	**100.0**	90.6	51.3
	Transformer (Relative Positional Encodings)	**100.0**	96.0	68.4
	Transformer (Only Positional Masking)	**100.0**	**100.0**	**99.8**
Boolean Expressions (5)	**LSTM (Baseline)**	**100.0**	99.5	96.0
	Transformer (Absolute Positional Encodings)	**100.0**	84.3	40.8
	Transformer (Relative Positional Encodings)	**100.0**	72.3	32.3
	Transformer (Only Positional Masking)	**100.0**	**99.8**	**99.0**
$a^n b^n$	**LSTM (Baseline)**	**100.0**	**100.0**	99.9
	Transformer (Absolute Positional Encodings)	**100.0**	**100.0**	**100.0**
	Transformer (Relative Positional Encodings)	**100.0**	**100.0**	**100.0**
	Transformer (Only Positional Masking)	**100.0**	**100.0**	**100.0**
$a^n b^n c^n$	**LSTM (Baseline)**	**100.0**	**100.0**	97.8
	Transformer (Absolute Positional Encodings)	**100.0**	62.1	5.3
	Transformer (Relative Positional Encodings)	**100.0**	31.3	22.0
	Transformer (Only Positional Masking)	**100.0**	**100.0**	**100.0**
$a^n b^n c^n d^n$	**LSTM (Baseline)**	**100.0**	**100.0**	99.9
	Transformer (Absolute Positional Encodings)	88.45	0.0	0.0
	Transformer (Relative Positional Encodings)	41.1	0.0	0.0
	Transformer (Only Positional Masking)	**100.0**	**100.0**	99.4

Table 8: The performance of Transformers and LSTMs on the respective counter languages. Refer to section 6 in the main paper for details.



(a) (b)

Figure 4: Attention maps for models trained on Shuffle-2 and Boolean-3 languages. Similar to our constructions for recognizing these languages, we observe nearly uniform attention weights in both cases

Language	Property	Dot-Depth	Transformer (Only Positional Masking) Bin 0	Transformer (Only Positional Masking) Bin 1	Transformer (w Position Encodings) Bin 0	Transformer (w Position Encodings) Bin 1	LSTM Bin 0	LSTM Bin 1
Tomita 1	SF	1	100.0	100.0	100.0	100.0	100.0	100.0
Tomita 4	SF (LT-k)	1	24.1	0.2	100.0	92.4	100.0	100.0
Tomita 7	SF	1	100.0	100.0	99.9	99.8	100.0	100.0
Tomita 2 = $\mathcal{D}_1 = (01)^*$	SF	1	100.0	100.0	100.0	100.0	100.0	100.0
$aa^*bb^*cc^*dd^*ee^*$	SF	1	100.0	100.0	100.0	100.0	100.0	100.0
$\{a,b\}^*d\{b,c\}^*$	SF	1	100.0	100.0	100.0	100.0	100.0	100.0
$\{0,1,2\}^*02^*$	SF	2	74.2	35.6	100.0	68.7	100.0	100.0
$\mathcal{D}_2$	SF	2	7.8	0.4	74.6	3.1	100.0	100.0
$\mathcal{D}_3$	SF	3	16.2	4.2	80.9	8.5	100.0	100.0
$\mathcal{D}_4$	SF	4	36.9	15.6	90.2	3.3	100.0	100.0
$\mathcal{D}_{12}$	SF	12	16.5	0.0	95.8	1.5	100.0	100.0
Parity	non-SF	–	22.0	0.0	68.7	0.0	100.0	100.0
$(aa)^*$	non-SF	–	0.0	0.0	100.0	0.0	100.0	100.0
$(aaaa)^*$	non-SF	–	0.0	0.0	100.0	0.0	100.0	100.0
$(abab)^*$	non-SF	–	0.0	0.0	100.0	2.5	100.0	100.0
Tomita 3	non-SF	–	9.8	9.8	75.4	10.8	100.0	100.0
Tomita 5	non-SF	–	4.9	0.0	29.3	0.0	100.0	100.0
Tomita 6	non-SF	–	9.1	0.0	88.8	0.0	100.0	100.0

Table 9: Summary of results on Regular Languages. The languages are arranged in an increasing order of their complexities.

Figure 5: Values of four different coordinates of the output of self-attention block. The model is trained to recognize Shuffle-4. The dotted lines are the scaled depth to length ratio for the four types of bracket provided for reference.

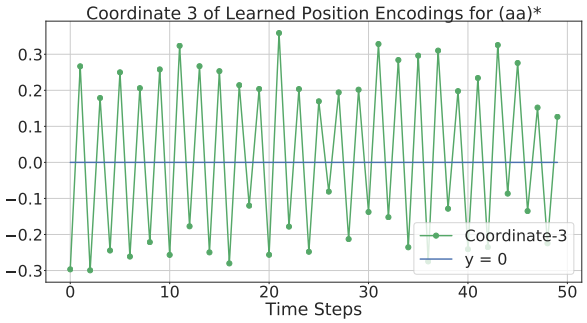


Figure 6: The values of coordiante 3 of the learned position encodings on the language $(aa)^*$. The variation in the encodings resemble a periodic behaviour similar to $cos(n\pi)$