Formal Language Learning with Transformers: Generalization, Counting, and Stack Augmentation

SHUNQI WANG, University of Toronto, Canada

Abstract

The Transformer architecture, which powers popular models like GPT (Generative Pretrained Transformer), has achieved remarkable success across a wide range of natural language processing tasks. However, despite its empirical performance, it struggles to generalize on seemingly simple formal language tasks—such as reversing a string or checking parity. In this paper, we first explain the self-attention mechanism and the standard (vanilla) Transformer architecture. We then introduce an augmented variant that incorporates a differentiable stack layer. To evaluate their expressive power, we test both the vanilla Transformer and the stack-augmented Transformer on a suite of tasks from the Chomsky hierarchy, including regular, deterministic context-free (DCF), counter, and context-sensitive languages. Our results show that under Optimal Hyperparameter setting, the stack-augmented Transformer outperforms the vanilla version on all counter language tasks and achieves perfect generalization on two out of four DCF tasks. Nevertheless, even with stack augmentation, the Transformer fails to generalize on certain regular languages, highlighting fundamental limitations in its inductive bias.

Additional Key Words and Phrases: counter languages, Transformers, generalization, inductive bias, stack

ACM Reference Format:

Shunqi Wang. 2025. Formal Language Learning with Transformers: Generalization, Counting, and Stack Augmentation. *ACM Trans. Graph.* 37, 4, Article 111 (August 2025), 12 pages. https://doi.org/XXXXXXX.XXXXXXX

1 Introduction

The Transformer architecture, built on self-attention layers [Vaswani et al. 2017], currently defines the state of the art in natural language processing. However, its ability to recognize languages defined in the Chomsky hierarchy remains largely unknown. While Transformers are theoretically Turing complete—under assumptions of infinite precision and unbounded memory tape [Pérez et al. 2021], their practical expressive power is much more constrained.

With appropriate positional encoding and hyperparameter tuning, Transformers can achieve non-trivial accuracy on tasks such as recognizing boolean expressions (a counter-type task) [Bhattamishra et al. 2020] and bucket sort (context-sensitive). Yet they consistently fail to generalize in tasks considered easier—such as all deterministic context-free (DCF) languages or even parity checking (a regular language) [Delétang et al. 2023b].

Author's Contact Information: Shunqi Wang, University of Toronto, Mississauga, Canada, shunqi.wang@mail.utoronto.ca.

ACM 1557-7368/2025/8-ART111
https://doi.org/XXXXXXX.XXXXXXX

One promising enhancement is to augment the Transformer with an extra layer that simulates stack behavior [Li et al. 2024]. The stack-augmented Transformer, with the stack-attention mechanism, has demonstrated the ability to recognize certain context-free languages—such as reverse string and simple stack manipulation. However, the original evaluation focused only on DCF tasks. We propose to extend this analysis further within the Chomsky hierarchy, particularly evaluating the model on counter languages like the Dyck-n language—i.e., balanced parentheses of maximum nesting depth n.

It is well-established that the standard Transformer [Vaswani et al. 2017] cannot generalize Dyck-n languages under fixed architecture constraints [Hahn 2020]. Empirically, self-attention models fail to generalize to deeper nesting in Dyck-n languages unless they include special tokens or architectural modifications [Ebrahimi et al. 2020]. Thus, it remains to be seen whether stack-attention augmentation can enable generalization on Dyck-n and even shuffle-n languages, which will measure the Transformer's capacity for learning counting behavior.

2 Related Work

Vaswani et al. [Vaswani et al. 2017] introduced the Transformer and its self-attention mechanism, replacing recurrence entirely. Their "Attention Is All You Need" architecture laid the groundwork for all subsequent advances in memory-augmented and language-modeling capabilities, demonstrating unprecedented performance on English–German translation.

Delétang et al. [Delétang et al. 2023b] present a comprehensive empirical study of RNNs, LSTMs, GRUs, and Transformers on tasks spanning the Chomsky hierarchy. They show that, even with exhaustive hyperparameter tuning, vanilla Transformers can handle certain regular and context-sensitive tasks but fail on deterministic context-free languages unless equipped with additional machinery. Notably, they push evaluation lengths up to 500 tokens—far beyond the training length of 40—highlighting severe generalization gaps.

Bhattamishra et al. [Bhattamishra et al. 2020] analyze both theoretical and empirical limitations of the standard Transformer for formal-language recognition. Rather than framing it as classification, they evaluate a character-prediction task: at each time step t, the model predicts a set of possible next characters, and is correct if all the true character is in that set. The true character means the next character that make the string still in the language. Under this stricter metric (zero-chance accuracy vs. ∼50% for classification), they claimed Transformers cannot generalize on non-star-free or Dyck-n languages beyond training depths.

DuSell and Chiang [DuSell and Chiang 2024] introduce a nondeterministic stack-attention mechanism in which each Transformer layer maintains a distribution over push/pop operations. This augmentation enables recognition of nontrivial context-free languages—such as balanced parentheses and reversed strings—that elude the vanilla

model. They report perfect generalization on all deterministic context-free tasks, albeit with substantial computational overhead.

DuSell and Chiang [DuSell and Chiang 2020] earlier showed that RNNs augmented with nondeterministic stacks can learn context-free languages. In their CoNLL 2020 paper, "Learning Context-Free Languages with Nondeterministic Stack RNNs," they achieve strong generalization on Dyck-n, outperforming traditional RNNs.

3 Preliminaries

In this section, we provide necessary technical background in understanding our work. We first go over the definition of the formal languages, then the self-attention and the vanilla Transformer, and the innovative stack attention[Li et al. 2024].

3.1 Formal Language and Chomsky Hierarchy

3.1.1 Formal Languages. A formal language is a set of finite strings built over a fixed finite alphabet Σ of symbols. For example, if $\Sigma = \{0, 1\}$, then the language

$$L = \{\, 0^n 1^n \mid n > 0\}$$

consists of all strings of n zeros followed by n ones. Grammars provide a finite, rule-based mechanism for generating exactly those strings: From a start symbol and apply production rules (rewritings of nonterminals to terminals and/or other nonterminals) until only terminals remain. The set of all derivable terminal strings is the grammar's formal language.

3.1.2 The Chomsky Hierarchy. The Chomsky hierarchy is a four-level classification of formal grammars (and the languages they generate) introduced by Noam Chomsky in 1956 [Chomsky 1956]. At the bottom are **Type 3 (regular) grammars**, whose languages can be recognized by finite-state automata. One level up are **Type 2 (context-free) grammars**, generating languages accepted by push-down automata(PDA) equipped with a stack.

Type 1 (context-sensitive) grammars sit above these, requiring a linear-bounded automaton (a Turing machine restricted to a tape whose size is proportional to the input). At the top are **Type 0 (recursively enumerable) grammars**, which correspond exactly to unrestricted Turing machines and can generate any computably enumerable language. Each higher level strictly covers the one below it, yielding the strict containment chain: Regular $\subset$ CF $\subset$ CS $\subset$ RE.

3.1.3 Counter Languages. Counter languages lie strictly between the context-free and context-sensitive classes in the Chomsky hierarchy. They are recognized by a **deterministic counter automaton** (DCA)−−a finite-state automaton augmented with one or more unbounded integer counters, each of which may be incremented, decremented by fixed constants, or reset to zero [J.Fischer and O.Rabin 1968]. Despite their finite control, DCAs can count arbitrarily high, allowing them to recognize non-context-free patterns such as

- $\{\, a^n b^n c^n \mid n > 0\}$,
- the language of palindromes of the form $\{\, ww^R \mid w \in \{a, b\}^*\}$,
- and other multiple-counting patterns.

Because they extend deterministic finite automata with simple numerical memory, counter automata can be seen as a restricted form of push-down automata (with a single stack symbol) or as finite automata equipped with integer counters instead of stacks. As a result, counter languages strictly contain all context-free languages yet remain a proper subset of the context-sensitive class. For instance, Counter automata can recognize the language $\{a^n b^n c^n \mid n > 0\}$ by maintaining two independent integer counters, whereas a PDA cannot achieve this. A PDA's stack can only be used once—after popping off the markers used to compare a's to b's, no memory remains to compare against c's.

3.1.4 Recognizing a Language. To **recognize** a language means to design an automaton or algorithm that—given any input string—will **accept** precisely those strings in the language and **reject** all others. For a regular language, this might be a finite-state automaton that changes state on each input symbol and ends in an accepting state exactly for strings in the language.

3.2 Standard Transformer Architecture

3.2.1 Self-Attention. Self-attention is the cornerstone of the vanilla Transformer [Vaswani et al. 2017]. A common instantiation of this mechanism is self-attention [Cheng et al. 2016; Parikh et al. 2016], a differentiable function that maps an input sequence of token representations $H = [h_1, \ldots, h_n] \in \mathbb{R}^{d\times n}$ to an output of the same shape. At each position i, the output is computed by attending from token i to all other tokens.

First, for each pair of token representation h_i and h_j, **compatibility scores**(Also known as attention scores) are computed via a scaled dot product, measuring their relevance and how much token j should the token i attend to:

$$e_{ij} = \frac{h_i^\top h_j}{\sqrt{d}}.$$

After all the compatibility scores have computed, these scores are normalized with a softmax over j, yielding the **self-attention distribution**(Also known as attention weight) for token i.

$$\alpha_i(j) = \frac{\exp(e_{ij})}{\sum_{k=1}^{n} \exp(e_{ik})},$$

Finally, to create the output for position i, we sum over the attention distribution that involves token i with the representations of the other inputs; the new representation at i is then

$$\mathrm{A}(H)_{:i} \;=\; \sum_{j=1}^{n} \alpha_i(j)\, h_j,$$

The mapping $\mathrm{A} : \mathbb{R}^{d\times n} \longrightarrow \mathbb{R}^{d\times n}$ defined by these operations is called an *attention head*. Given H, the head computes compatibility scores e_{ij}, normalizes them into weights $\alpha_i(j)$, and produces each output column $A(H)_{:i}$ by summing the weighted inputs.

Crucially, because the computations for all positions run in parallel, self-attention is permutation-invariant over the input, which accelerates both training and inference. In multi-head attention, this process is repeated in parallel across several heads, allowing the model to attend to different representation subspaces simultaneously. Moreover, A is fully differentiable, enabling its parameters to be learned via gradient descent.

3.2.2 Queries, Keys, and Values. In the version of the attention mechanism introduced by Vaswani [Vaswani et al. 2017], before computing the dot-product attention scores, each input vector $h_i \in \mathbb{R}^d$ is linearly projected into three distinct spaces:

$$q_i = W_Q\, h_i, \quad k_i = W_K\, h_i, \quad v_i = W_V\, h_i,$$

where $W_Q, W_K, W_V \in \mathbb{R}^{d\times d}$ are learned parameter matrices. The *query* q_i represents the "question" at position i, the *key* k_j encodes the "address" of position j, and the *value* v_j carries the information to be aggregated. We compute compatibility scores via a scaled dot product between queries and keys:

$$e_{ij} = \frac{q_i^\top k_j}{\sqrt{d}}.$$

These scores are normalized with softmax over j to get the attention distribution for token i over token j.

$$\alpha_i(j) = \frac{\exp(e_{ij})}{\sum_{k=1}^{n}\exp(e_{ik})},$$

The final output at position i is the sum of the product of the attention distribution and the corresponding value vector.

$$\mathrm{A}(H)_{:i} \;=\; \sum_{j=1}^{n} \alpha_i(j)\, v_j.$$

This separation into queries, keys, and values allows the model to learn distinct projections for matching (query–key) and content aggregation (value), improving flexibility and representation power.

3.2.3 Transformer Architecture. Let the input be a sequence of N tokens $w = (w_1, \ldots, w_N)$. We denote by d the model dimension and by L the number of Transformer layers. The Transformer computes an output matrix $H^L \in \mathbb{R}^{d\times N}$ through the following steps:

Step 0: Embedding & Positional Encoding. Each token w_i is mapped to a d-dimensional embedding and augmented with a positional encoding (Positional encoding is optional):

$$H^0 = \big[\,\mathrm{Embed}(w_1), \ldots, \mathrm{Embed}(w_N)\big] \;+\; \mathrm{PE} \;\in\; \mathbb{R}^{d\times N},$$

If positional encoding is enabled, PE will inject position information, breaking permutation invariance, which is indispensable for the order-sensitive tasks.

Step k ($1 \le k \le L$): Stacked Transformer Layers. Each layer consists of two sub-layers with residual connections and layer normalization (LN):

1. **Multi-Head Self-Attention (MHA)**

$$H^l_{MHA} = \mathrm{LN}\Big(H^{l-1} + \mathrm{MHA}\big(H^{l-1}\big)\Big) \quad \in\; \mathbb{R}^{d\times N}.$$

2. **Position-wise Feed-Forward Network (FFN)**

$$H^l = \mathrm{LN}\Big(H^l_{MHA} + \mathrm{FFN}\big(H^l_{MHA}\big)\Big) \quad \in\; \mathbb{R}^{d\times N}.$$

Here:

- $\mathrm{MHA}(\cdot)$ applies multiple parallel scaled-dot-product attention heads (Queries/Keys/Values) and concatenates their outputs.
- $\mathrm{FFN}(X) = W_2\,\mathrm{ReLU}(W_1 X + b_1) + b_2$ is a feed-forward network applied independently at each position.
- Residual connections ('+') wrap each sub-layer, and $\mathrm{LN}(\cdot)$ denotes layer normalization.
- There is no parallelism across layers, layer H^l input depends on the output from previous layer H^{l-1}. Thus, increasing Transformer's layers dramatically rises the training time.

Final Output. After L layers, the Transformer outputs

$$H^L \;=\; \mathrm{Transformer}(w) \;\in\; \mathbb{R}^{d\times N},$$

which can then be passed to task-specific heads (e.g., classification, sequence generation).

3.3 Stack Attention Mechanism

We follow the stack-attention design of Li et al. [Li et al. 2024], augmenting each Transformer layer with a differentiable "stack" over input positions.

3.3.1 Stack structures and Operations. We use the terms *timestep* and *position* interchangeably.

1. Stack Initialization. Introduce a special [BOS] token ("Beginning Of Sequence") at position 0. This token represents the *empty* stack. For each position $i \in \{0, \ldots, N\}$, we maintain a vector $\alpha_i \in \mathbb{R}^{N+1}$, which is a probability distribution over indices $0, \ldots, N$. For example, $\alpha_1 = [0, 1, 0, 0]$ means that at position 1, the stack "points to" input index 1. At position 0, initialize the stack as

$$\alpha_0 := [1, 0, 0, \ldots, 0]^\top \in \mathbb{R}^{N+1},$$

so that it points only to the [BOS] symbol (empty stack).

2. Accessing Stack Elements. We write $\alpha_i(j)$ for the $(j+1)$th entry of α_i. In particular, $\alpha_i(0)$ is the first entry, $\alpha_i(1)$ is the second, …

3. Stack Operations. At each subsequent position i, the new stack state α_i is computed from α_{i-1} plus the combination of the following three operations:

$$\alpha_i^{\mathrm{PUSH}}(j) = \begin{cases} 1, & j = i, \\ 0, & \text{otherwise}, \end{cases} \tag{a}$$

$$\alpha_i^{\text{NO-OP}} = \alpha_{i-1}, \tag{b}$$

$$\alpha_i^{\mathrm{POP}} = \sum_{j=1}^{i-1} \alpha_{i-1}(j)\,\alpha_{j-1}. \tag{c}$$

PUSH gives a one-hot vector pointing to the current index of the input being processed. NO-OP simply copy the previous stack. POP removes the top element, backtracks to the second element in the stack.

4. Compute stack attention for position i from action weights. Let $a_i = \big[a_i^{\mathrm{PUSH}}, a_i^{\mathrm{POP}}, a_i^{\text{NO-OP}}\big] \in \Delta^2$ be the mixture weights over the three operations, computed by $a_i = \mathrm{softmax}\big(W_a\, h_i + b_a\big)$, where $h_i \in \mathbb{R}^D$ is the transformer hidden state at position i, $W_a \in \mathbb{R}^{3\times D}$ and $b_a \in \mathbb{R}^3$ are learnable. The resulting stack attention at i is

$$\alpha_i = a_i^{\mathrm{PUSH}}\,\alpha_i^{\mathrm{PUSH}} + a_i^{\mathrm{POP}}\,\alpha_i^{\mathrm{POP}} + a_i^{\text{NO-OP}}\,\alpha_i^{\text{NO-OP}}.$$

5. Compute weighted sum for all positions. Given the matrix of hidden states $H = [\, h_0; h_1; \ldots; h_N] \in \mathbb{R}^{(N+1)\times D}$, we compute the stack-based context vector at position i as

$$S(H)_{:,i} = \sum_{n=0}^{N} \alpha_i(n)\, h_n.$$

Thus $S(H) \in \mathbb{R}^{(N+1)\times D}$ parallels standard self-attention, but uses the stack-derived weights $\{\alpha_i\}$.

3.3.2 One Iteration of Stack Attention. Given the input embedding matrix $H = \big[h_0; h_1; \ldots; h_N\big] \in \mathbb{R}^{(N+1)\times D}$, we compute the stack-augmented context at position i via the function S as follows:

(1) Take the hidden state h_i (the ith row of H).

(2) Compute soft action weights a_i for timestamp i, i.e. the distribution over stack actions (push, pop, no-op). Note that the stack is soft. We compute a_i using the hidden state h_i and a learned weight matrix W_a:

$$a_i = \text{softmax}\big(W_a\, h_i + b_a\big) = \big[a_i^{\text{PUSH}},\ a_i^{\text{POP}},\ a_i^{\text{NO-OP}}\big]. \tag{1}$$

(3) Meanwhile, simulate each stack operation using the previous stack state α_{i-1} to obtain α_i^{PUSH}, α_i^{POP}, and $\alpha_i^{\text{NO-OP}}$.

(4) Combine the action weights and the new stack states to get the stack attention for position i:

$$\alpha_i = a_i^{\text{PUSH}}\, \alpha_i^{\text{PUSH}} + a_i^{\text{POP}}\, \alpha_i^{\text{POP}} + a_i^{\text{NO-OP}}\, \alpha_i^{\text{NO-OP}}. \tag{2}$$

Each entry $\alpha_i(n)$ tells "how much position i is focusing on position n."

(5) Lastly, use the stack attention weights to compute the output at position i by averaging past hidden states, similarly to standard scaled-dot attention:

$$S(H)_{:,i} = \sum_{n=0}^{N} \alpha_i(n)\, h_n. \tag{3}$$

This has the same form as standard self-attention's value-weighted sum, but α_i is computed via simulated stack operations instead of dot-product scores.

3.3.3 Transformer inserted with stack attention. The Stack Transformer modifies each standard Transformer layer by inserting a third sub-layer, the attention mechanism for layer ℓ becomes:

(1) **Multi-Head Self-Attention:**

$$H^{M(\ell)} = \text{LN}\big(\text{MHA}(H^{(\ell-1)}) + H^{(\ell-1)}\big).$$

(2) **Feedforward Network:**

$$H^{F(\ell)} = \text{LN}\big(\text{FFN}(H^{M(\ell)}) + H^{M(\ell)}\big).$$

(3) **Stack Attention Mechanism (New Sub-layer):**

$$H^{S(\ell)} := S\big(H^{F(\ell)}\big) + H^{F(\ell)}, \tag{4}$$

$$H^{(\ell)} := H^{S(\ell)}.(\text{Final output of layer } \ell) \tag{5}$$

As with other sub-layers, we apply a residual connection by adding the output of the stack attention mechanism S, to its input, enabling the model to skip the stack when necessary.

4 Methods

4.1 Experiment Setup

We follow a similar experimental procedure to that of Delétang et al. [Delétang et al. 2023b]. For the Transformer model, we use 5 layers, 8 attention heads, and a model dimension of 64. For each task, the model is trained for 100,000 steps, which we found to be sufficient for convergence in all cases.

Length generalization has been a central focus in this line of research [Delétang et al. 2023a; Joulin and Mikolov 2015]. To evaluate the model's ability to generalize to longer sequences, we train it on sequences of length 0–40 and test on sequences of length 40–100.

Furthermore, to assess the positional awareness of models with stack augmentation, we deliberately use only the Transformer encoder in our experiments. This choice removes the effect of future masking (which provides some implicit positional information) and isolates the model's ability to handle positional reasoning.

4.2 Architectures

The architectures we experimented with include the vanilla Transformer, the Transformer with stack attention, and the Stack-RNN. The Stack-RNN [Joulin and Mikolov 2015] can perform any linear combination of PUSH, POP, and NO-OP operations on the stack. It achieves the highest accuracy in all experiments and serves as the baseline for comparison.

4.3 Hyperparameters Explored

The Hyperparameters we tuned include the learning rate and positional encoding. Our experiments cover a wide range of learning rates, including **1e-3, 5e-4, 1e-4, and 5e-5**. We will present the findings related to learning rate patterns separately in the results section.

For positional encoding, there are only two choices: either not using positional encoding or using ALiBi encoding. The main reason is that the authors of Stack Attention [Li et al. 2024] claimed that most positional encodings do not appreciably improve the performance of Transformers equipped with stack attention and may even degrade it. Therefore, it is not worthwhile for us to experiment with various types of positional encoding. Since the authors identified ALiBi as providing the most significant improvement, we choose to use only that one.

4.4 Regular, DCF, and CS Tasks

We adopt the same task framework as Delétang [Delétang et al. 2023b], grouping problems into three classes:

- **Regular** (finite-state)
- **Deterministic Context-Free (DCF)**
- **Context-Sensitive (CS)**

Language Transduction Setup. Rather than training a recognizer, we learn a *transducer* f that maps each input string $x \in \Sigma_I^*$ to an output string $y \in \Sigma_O^*$ (Σ_I is the input language, Σ_O is the output language):

$$f : \Sigma_I^* \longrightarrow \Sigma_O^*.$$

To cast this as a language-recognition problem, we build the combined language

$$L_f \;=\; \left\{ x\, f(x) \;\middle|\; x \in \Sigma_I^* \right\} \;\subseteq\; (\Sigma_I \cup \Sigma_O)^*.$$

A machine recognizing L_f must (1) check that x is a valid input word (Membership of x in $x \in \Sigma_I^*$), and (2) verify that what follows matches the correct output $f(x)$. That is, the model's prediction y given the input x, matches the true output x-$f(x)$.

Alphabets. We use the same alphabets Σ_I and Σ_O for each task category as in Delétang[Delétang et al. 2023b], ensuring a one-to-one comparison across Regular, DCF, and CS transduction tasks.

For the detailed descriptions of each task, we include those on which the stack-augmented Transformer performed well, which can be found in Appendix A. Descriptions for the remaining tasks can be found in Delétang's paper [Delétang et al. 2023b].

4.5 Counter Language Tasks

Different from the strict setting in the work of Bhattamishra [Bhattamishra et al. 2020], our counter language tasks are defined by mapping sequences containing brackets to the set of sequences that satisfy the corresponding counter language. To thoroughly evaluate the model, we design meaningful negative examples instead of generating them randomly.

Our counter languages include Dyck-1, Shuffle-2, Shuffle-4, and Shuffle-6. To recognize and generalize well on these tasks, the model needs to learn the underlying counting behavior. We describe some of our task setting up below:

4.5.1 Dyck-1. The Dyck-1 language consists of well-formed strings using only one type of bracket, namely '(' and ')'. In this task, we perform *language recognition via binary transduction*, where the goal is to determine whether an input string belongs to the Dyck-1 language.

- **Input language** Σ_I: Arbitrary sequences over '(' and ')', which may be unbalanced or ill-formed.
- **Output language** Σ_O: A binary label indicating whether the input is a well-formed Dyck-1 sequence.

To standardize the output:

- Output `0` if the input belongs to the Dyck-1 language (i.e., the parentheses are balanced),
- Output `1` otherwise (i.e., the parentheses are unbalanced).

- **Positive Example**: Input – `()()(())`
  Output – `0` (balanced)
- **Negative Example**: Input – `(()(((`
  Output – `1` (unbalanced)

To make the task sufficiently challenging, we do not generate negative examples by randomly sampling bracket sequences. Instead, for the Dyck-1 language, we consider a specific and minimal form of corruption:

- We swap exactly one pair of brackets, resulting in a sequence like `)(()`. While the total number of opening and closing brackets remains correct, the order is invalid, making the sequence unbalanced.

This design is particularly interesting because the error lies solely in the order of the brackets. As a result, a vanilla Transformer encoder without positional encoding cannot distinguish such patterns, as we demonstrate later in the Result section.

4.5.2 Shuffle-2. The Shuffle-2 language consists of well-formed strings containing exactly two types of brackets, such as parenthesis '( )' and curly bracket '[ ]'. Formally, the language is defined as the shuffle of two Dyck-1 languages: one over the bracket type '( )' and the other over '[ ]'.

- Output `0` if the input belongs to the Shuffle-2 language (i.e., all types of brackets are balanced),
- Output `1` otherwise (i.e., at least one type of bracket is unbalanced).

- **Positive Example**: Input – `[()(]()[][])`
  Output – `0` (balanced)
- **Negative Example**: Input – `(][`
  Output – `1` (unbalanced)

To make the task sufficiently challenging, we do not generate negative examples by randomly sampling bracket sequences. Instead, starting from a valid example, we apply two different perturbations to create structured negative examples:

- **Option 1**: Swap exactly one pair of brackets, yielding a sequence like `(][)`. Although the total number of brackets is preserved, the ordering is incorrect.
- **Option 2**: Break exactly one bracket by altering its type. For example, replacing a '[' with a '(' results in `()[)`. Detecting such corruption requires the model to distinguish between bracket types correctly.

For all the shuffle and Dyck language, we design the sample batch consists equal number of positive and negative examples. Another important characteristic of the Shuffle language we implemented is that the number of bracket pairs for each type (e.g., curly brackets and parentheses) is not necessarily balanced with respect to each other. That is, we randomize the number of pairs for each bracket type independently when generating examples in the Shuffle language.

For the Shuffle-4 and Shuffle-6 languages, since they follow the same structure as Shuffle-2 but involve the shuffle of more Dyck-1 sequences (i.e., more bracket types), we do not provide additional explanation for them.

5 Results and Discussions

We evaluate all the architectures described in Section 4 on the full set of tasks. Results are organized into separate sections based on the task complexity level in the Chomsky hierarchy.

For each task, we report the **average accuracy** over all testing samples with input lengths ranging from 1 to 100. After presenting the results for each task category, we provide a discussion and analyze potential factors contributing to the observed performance.

5.1 Regular Tasks

The results for regular tasks are presented in the Table 1, for all architectures.

Table 1. Results for regular language recognition tasks. Both architectures are tuned with the best Hyperparameters. The accuracy reported is the average accuracy across all samples. *CN*: Cycle Navigation, *EP*: Even Pairs, *MA*: Modular Arithmetic(Simple), *PC*: Parity Checks

Task	Model	PE	Learning Rate	Accuracy
CN	Stack RNN	None	1e-3	1.0000
	Vanilla Transformer	None	5e-4	0.8639
	Stack Transformer	None	5e-4	0.7247
	Vanilla Transformer	ALiBi	5e-5	0.8017
	Stack Transformer	ALiBi	5e-4	0.9073
EP	Stack RNN	None	1e-3	1.0000
	Vanilla Transformer	None	5e-5	0.5260
	Stack Transformer	None	5e-4	1.0000
	Vanilla Transformer	ALiBi	1e-4	0.9585
	Stack Transformer	ALiBi	5e-4	1.0000
MA	Stack RNN	None	1e-3	1.0000
	Vanilla Transformer	None	1e-4	0.2366
	Stack Transformer	None	1e-4	0.4125
	Vanilla Transformer	ALiBi	1e-4	0.3460
	Stack Transformer	ALiBi	1e-4	0.3494
PC	Stack RNN	None	1e-3	1.0000
	Vanilla Transformer	None	5e-5	0.7046
	Stack Transformer	None	5e-4	0.6989
	Vanilla Transformer	ALiBi	1e-4	0.6866
	Stack Transformer	ALiBi	5e-5	0.6360

Table 2. Results for DCF-language recognition tasks. Both architectures are tuned with the best Hyperparameters. The accuracy reported is the average accuracy across all samples. *MAB*: Modular Arithmetic w/ Brackets, *RS*: Reverse String, *SE*: Solve Equation, *SM*: Stack Manipulation

Task	Model	PE	Learning Rate	Accuracy
MAB	Stack RNN	None	1e-3	0.8319
	Vanilla Transformer	None	1e-4	0.3461
	Stack Transformer	None	1e-4	0.3755
	Vanilla Transformer	ALiBi	5e-4	0.4202
	Stack Transformer	ALiBi	5e-4	0.4210
RS	Stack RNN	None	1e-3	1.0000
	Vanilla Transformer	None	5e-4	0.5682
	Stack Transformer	None	5e-4	1.0000
	Vanilla Transformer	ALiBi	5e-4	0.8674
	Stack Transformer	ALiBi	5e-4	1.0000
SE	Stack RNN	None	1e-3	0.9132
	Vanilla Transformer	None	1e-4	0.2734
	Stack Transformer	None	1e-4	0.3084
	Vanilla Transformer	ALiBi	1e-4	0.3643
	Stack Transformer	ALiBi	1e-4	0.3651
SM	Stack RNN	None	5e-4	1.0000
	Vanilla Transformer	None	1e-4	0.5022
	Stack Transformer	None	5e-4	0.9975
	Vanilla Transformer	ALiBi	1e-3	0.8269
	Stack Transformer	ALiBi	5e-4	0.9959

As expected, the Stack RNN achieves perfect accuracy on all regular-language tasks, exactly reproducing the results reported by Delétang etal.[Delétang et al. 2023b]. When we compare the stack-augmented Transformer to the vanilla Transformer, the stack variant clearly outperforms on both the Cycle Navigation and Even Pairs tasks. In particular, the stack Transformer reaches perfect accuracy at every test length on Even Pairs—despite this task depending only on matching the first and last symbols of the input rather than computing a true parity (see Appendix A).

Nevertheless, even with an explicit stack, the Transformer fails to recognize certain regular languages such as Parity Check, plateauing around 70% accuracy (versus 50% chance). Li etal.[Li et al. 2024] argue that positional encodings degrade performance on DCF tasks, yet we observe that adding ALiBi actually improves the stack Transformer's performance on Cycle Navigation. Overall, since the stack-augmented Transformer cannot solve all regular recognition tasks, we conjecture that it still lacks the full expressivity of a finite-state automaton.

5.2 DCF Tasks

The results for DCF Tasks are presented in the Table 2.

The Stack RNN continues to achieve near-perfect accuracy on all DCF tasks, confirming that it effectively emulates the behavior of a push-down automaton (PDA), which is required for deterministic context-free recognition [Joulin and Mikolov 2015]. In agreement with Li et al. [Li et al. 2024], we observe that adding a stack to the Transformer improves performance on two of the four DCF tasks—Reverse String and Simulation—but yields little or no gain on the others. This makes intuitive sense: for Reverse String, the stack-augmented model can push each input symbol onto its stack in order, then pop them to produce the reversed output.

However, despite this enhancement, the stack Transformer still fails to solve every DCF task, indicating it does not fully capture the power of a true PDA. We hypothesize this limitation stems from its simplistic stack implementation, which records only token positions rather than their full embeddings. By contrast, DuSell and Chiang's nondeterministic stack attention model pushes the actual transformed input vectors onto its stack, and they report that it can recognize all context-free languages [DuSell and Chiang 2024].

5.3 CS Tasks

The results for CS tasks are presented in the Table 3.

Table 3. Results for context-sensitive language recognition tasks. *BA*: Binary Addition, *BM*: Binary Multiplication, *CS*: Compute Square Root, *DS*: Duplicate String, *OF*: Odds First, *BS*: Bucket Sort

Task	Model	PE	Learning Rate	Accuracy
BA	Stack RNN	None	5e-4	0.7676
	Vanilla Transformer	None	1e-3	0.6170
	Stack Transformer	None	5e-4	0.7977
	Vanilla Transformer	ALiBi	1e-3	0.7689
	Stack Transformer	ALiBi	5e-4	0.7523
BM	Stack RNN	None	1e-3	0.6510
	Vanilla Transformer	None	1e-3	0.5198
	Stack Transformer	None	5e-4	0.6654
	Vanilla Transformer	ALiBi	5e-4	0.6650
	Stack Transformer	ALiBi	5e-4	0.6634
CS	Stack RNN	None	1e-3	0.7489
	Vanilla Transformer	None	5e-5	0.5415
	Stack Transformer	None	1e-4	0.6747
	Vanilla Transformer	ALiBi	5e-4	0.6570
	Stack Transformer	ALiBi	1e-4	0.6624
DS	Stack RNN	None	1e-3	0.3997
	Vanilla Transformer	None	1e-3	0.2939
	Stack Transformer	None	5e-5	0.5766
	Vanilla Transformer	ALiBi	1e-4	0.5566
	Stack Transformer	ALiBi	5e-5	0.5766
OF	Stack RNN	None	1e-3	0.4792
	Vanilla Transformer	None	1e-3	0.2943
	Stack Transformer	None	5e-4	0.6669
	Vanilla Transformer	ALiBi	1e-3	0.5794
	Stack Transformer	ALiBi	5e-4	0.5868
BS	Stack RNN	None	1e-3	0.9846
	Vanilla Transformer	None	5e-4	0.2941
	Stack Transformer	None	1e-4	0.9883
	Vanilla Transformer	ALiBi	1e-3	0.8762
	Stack Transformer	ALiBi	5e-5	0.9339

For the context-sensitive tasks, the Stack RNN's accuracy drops sharply and is even outperformed by both Transformer variants in several cases. This decline is expected: a Stack RNN can only emulate a push-down automaton, which lacks the computational power needed for context-sensitive languages. Although neither the vanilla nor the stack-augmented Transformer achieves consistently high accuracy across CS tasks, both outperform the Stack RNN on Binary Multiplication and Duplicate String.

Notably, on Bucket Sort the vanilla Transformer reaches 87.6% accuracy, while the stack-augmented Transformer climbs to 98.8%. The behavior of sorting requires the model to maintain multiple independent counters to count the occurrence of each type of tokens, meaning it is a non-context-free operation. This result highlights how the stack attention mechanism can extend the Transformer's capability, enabling it to approximate more powerful behaviors such as sorting.

5.4 Tasks For Counter Languages

The results for Counter language tasks are presented in the Table 4, for all architectures.

Table 4. Results for counter (Dyck/Shuffles) language recognition tasks. Both architectures are tuned with the best Hyperparameters. The accuracy reported is the average accuracy across all samples. *DY*: Dyck, *S2*: Shuffle-2, *S4*: Shuffle-4, *S6*: Shuffle-6

Task	Model	PE	Learning Rate	Accuracy
DY	Stack RNN	None	1e-3	0.9975
	Vanilla Transformer	None	1e-4	0.6197
	Stack Transformer	None	1e-4	0.9996
	Vanilla Transformer	ALiBi	1e-4	0.9353
	Stack Transformer	ALiBi	5e-5	0.9617
S2	Stack RNN	None	1e-3	0.8622
	Vanilla Transformer	None	1e-4	0.5067
	Stack Transformer	None	1e-4	0.9002
	Vanilla Transformer	ALiBi	1e-4	0.8275
	Stack Transformer	ALiBi	5e-4	0.8299
S4	Stack RNN	None	5e-4	0.8950
	Vanilla Transformer	None	5e-5	0.5018
	Stack Transformer	None	1e-4	0.8708
	Vanilla Transformer	ALiBi	5e-4	0.8439
	Stack Transformer	ALiBi	1e-4	0.8411
S6	Stack RNN	None	5e-4	0.8970
	Vanilla Transformer	None	5e-5	0.5026
	Stack Transformer	None	5e-4	0.8220
	Vanilla Transformer	ALiBi	5e-4	0.8262
	Stack Transformer	ALiBi	5e-4	0.8420

The Counter languages occupy a position between DCF and CS, which explains why the Stack RNN—capable of simulating a PDA can still handle them effectively. Indeed, it achieves perfect accuracy on Dyck-1 and around 90% on all shuffle-k tasks, though the latter remains challenging even for this model.

By contrast, a vanilla Transformer without positional encodings fails outright: many negative examples differ only by swapping a single matching bracket pair, and a position-invariant encoder treats those inputs identically. Adding positional encodings dramatically improves performance—pushing accuracy above 80%—but the stack-augmented Transformer consistently outperforms it, demonstrating that an explicit stack is a more powerful inductive bias than positional encoding alone.

Interestingly, as we move from Dyck-1 to Shuffle-6, the stack-augmented Transformer's accuracy steadily declines. We conjecture that each bracket type effectively requires its own stack (Or at least, own counter) to track nesting, so the Transformer's single-stack mechanism becomes increasingly strained as the alphabet grows. Li et al.[Li et al. 2024] argued that even with a stack, a Transformer cannot recognize all context-free languages without positional encodings. Our experiments refine this picture: with an appropriate

learning rate and positional encoding, the stack-augmented Transformer can simulate simple counting behaviors and achieve non-trivial performance on counting languages, though it still falls short of full context-free expressivity.

5.5 Learning rate and accuracy

Generalization accuracy in deep learning is highly sensitive to the choice of learning rate, which is why most experimental studies treat it as a key hyperparameter. However, relatively few works systematically evaluate how varying the learning rate influences final performance. In this paper, we will display: Across all proposed architectures, how different learning rates impact generalization accuracy on the DCF tasks Reverse String and Stack Manipulation.

Table 5. Reverse String: Accuracy vs. Learning Rate.

Model (PE; Using Stack)	Learning Rate			
	1e-3	5e-4	1e-4	5e-5
Stack RNN (NONE; no)	1.0000	1.0000	1.0000	1.0000
Transformer (NONE; no)	0.5679	0.5682	0.5678	0.5669
Transformer (ALiBi; no)	0.8410	0.8674	0.8475	0.8376
Transformer (NONE; yes)	0.5676	1.0000	1.0000	1.0000
Transformer (ALiBi; yes)	0.9997	1.0000	1.0000	1.0000

Table 6. Stack Manipulation: Accuracy vs. Learning Rate.

Model (PE; Using Stack)	Learning Rate			
	1e-3	5e-4	1e-4	5e-5
Stack RNN (NONE; no)	0.9996	1.0000	1.0000	1.0000
Transformer (NONE; no)	0.5003	0.5019	0.5022	0.5015
Transformer (ALiBi; no)	0.8269	0.8109	0.8189	0.8045
Transformer (NONE; yes)	0.9917	0.9975	0.9459	0.9444
Transformer (ALiBi; yes)	0.2039	0.2607	0.3651	0.3296

We report the remaining learning-rate vs. accuracy curves in the Appendix. From the two tables above, we following observations:

- **Optimal learning rates vary by task.** Even with the same architecture and positional encoding, the best learning rate can differ dramatically across language tasks.
- **learning rate can affect accuracy a lot** Change of learning rate can bring some significant surge of accuracy. For example, the stack-augmented Transformer without PE achieves only 56% accuracy at a learning rate of 1e-3, yet reaches 100% accuracy at any lower rate. As such accuracy fluctuation is dominant, we infer the learning rate not only controls convergence speed but can determine whether and how well the model converges.
- **A single-peaked accuracy curve.** Whenever accuracy is not saturated at 100%, increasing the learning rate produces either (a) a rise followed by a fall, (b) a monotonic rise, or (c) a monotonic fall. We never observe a secondary increase after an initial decrease. In practice, this means there is a single optimal learning rate. To find it, one should look for the point where accuracy first climbs with increasing rate and then begins to drop. If no such peak appears, the search range must be extended.

6 Conclusions

In this work, we have explored the expressive power of the Transformer architecture within the Chomsky hierarchy, and evaluated the effect of a novel stack-attention augmentation. Our main contributions and findings are as follows:

- We reviewed the limitations of the vanilla Transformer on formal-language tasks, showing that—even with positional encodings—it fails to generalize on many deterministic context-free (DCF) and counter-language tasks beyond its training length.
- Follows [Li et al. 2024] we implemented the stack-augmented Transformer, which inserts a differentiable stack-attention sub-layer into each self-attention block, and compared it against the vanilla Transformer and a Stack-RNN baseline.
- Across DCF (e.g. Reverse String, Stack Manipulation), counter (Dyck-1, shuffle-k), and CS (e.g. bucket sort) tasks, the stack-augmented Transformer consistently outperforms the vanilla model, demonstrating both push–pop behavior and parallel counting capabilities that standard self-attention alone cannot achieve.
- We carried out a study of learning-rate sensitivity, observing that the deep learning model exhibits a single accuracy peak at an optimal rate, and that small changes can make the difference between complete failure and perfect generalization.

References

Satwik Bhattamishra, Kabir Ahuja, and Navin Goyal. 2020. On the Ability and Limitations of Transformers to Recognize Formal Languages. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP 2020)*. Association for Computational Linguistics, Online (Virtual), 7096–7116.

Jianpeng Cheng, Li Dong, and Mirella Lapata. 2016. Long Short-Term Memory-Networks for Machine Reading. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing (EMNLP 2016)*. Association for Computational Linguistics, 551–561.

Noam Chomsky. 1956. Three Models for the Description of Language. *IRE Transactions on Information Theory* 2, 3 (1956), 113–124. doi:10.1109/TIT.1956.1056813

Guillaume Delétang, Camille Chalumeau, Valentin Liévin, and Thomas Cohen. 2023a. Neural Networks Fail to Learn Simple String Transformations. *Transactions of the Association for Computational Linguistics (TACL)* (2023).

Grégoire Delétang, Anian Ruoss, Jordi Grau-Moya, Tim Genewein, Li Kevin Wenliang, Elliot Catt, Chris Cundy, Marcus Hutter, Shane Legg, Joel Veness, and Pedro A. Ortega. 2023b. Neural Networks and the Chomsky Hierarchy. In *Proceedings of the 11th International Conference on Learning Representations (ICLR)*. OpenReview.net, Kigali, Rwanda. Preprint originally available at arXiv:2207.02098.

Brian DuSell and David Chiang. 2020. Learning Context-Free Languages with Nondeterministic Stack RNNs. In *Proceedings of the 24th Conference on Computational Natural Language Learning (CoNLL 2020)*, Raquel Fernández and Tal Linzen (Eds.). Association for Computational Linguistics, Online, 507–519. doi:10.18653/v1/2020.conll-1.41

Brian DuSell and David Chiang. 2024. Stack Attention: Improving the Ability of Transformers to Model Hierarchical Patterns. In *The 12th International Conference on Learning Representations (ICLR) 2024*. OpenReview.net, Vienna, Austria. https://openreview.net/forum?id=XVhm3X8Fum

Javid Ebrahimi, Dhruv Gelda, and Wei Zhang. 2020. How Can Self-Attention Networks Recognize Dyck-n Languages?. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics, Online (Virtual), 4301–4306. doi:10.18653/v1/2020.findings-emnlp.384

Michael Hahn. 2020. Theoretical Limitations of Self-Attention in Neural Sequence Models. *Transactions of the Association for Computational Linguistics* 8 (2020), 156–171. doi:10.1162/tacl_a_00306

Michael J.Fischer and Michael O.Rabin. 1968. Super-Exponential Complexity of Presburger Arithmetic. *Proceedings of the SIAM–AMS Symposium on Applied Algebra* (1968), 27–41.

Armand Joulin and Tomas Mikolov. 2015. Inferring Algorithmic Patterns with Stack-Augmented Recurrent Nets. In *Advances in Neural Information Processing Systems (NeurIPS)*.

Jiaoda Li, Jennifer C. White, Mrinmaya Sachan, and Ryan Cotterell. 2024. A Transformer with Stack Attention. In *Findings of the Association for Computational Linguistics: NAACL 2024*. Association for Computational Linguistics, Mexico City, Mexico, 4318–4335. Augments each transformer layer with a deterministic stack-attention sublayer (arXiv:2405.04515).

Ankur P. Parikh, Oscar Täckström, Dipanjan Das, and Jakob Uszkoreit. 2016. A Decomposable Attention Model for Natural Language Inference. In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing (EMNLP 2016)*. Association for Computational Linguistics, 2249–2255.

Jorge Pérez, Pablo Barceló, and Javier Marinković. 2021. Attention Is Turing-Complete. *Journal of Machine Learning Research* 22 (2021), 75:1–75:35. doi:10.5555/3546258.3546333 Shows that Transformers with hard attention, arbitrary precision, and unbounded memory are Turing complete.

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, 30*. Curran Associates, Inc., Long Beach, CA, USA, 5998–6008. Originally published as arXiv:1706.03762.

A Tasks

Following the Task set up in Delétang's paper[Delétang et al. 2023b], in this section we describe the task we experiment with in more details.

Even Pairs (Regular). Let x be a binary string (e.g.*aabba*). We say x has an *even pair* if the total number of occurrences of the substrings ab and ba is even. For example, in *aabba* there is one ab and one ba, for a total of two, which is even; we therefore assign output $y = \mathsf{b}$ (interpreting "even" as b). Observing that counting ab and ba pairs reduces to checking whether the first and last symbols of the string are equal, we see that this language is regular. A minimal DFA with two states suffices: it records the first symbol on entry and, upon reading the final symbol, compares it to the stored symbol to decide acceptance. *Note:* This is not a parity-checking task over all bits, but rather a simple "first-equals-last" test.

Cycle Navigation (Regular). This task involves determining an agent's final position on a cycle of length 5, given a sequence of movements encoded by the alphabet $\{0, 1, 2\}$, where 0 denotes STAY, 1 denotes INCREASE by 1, and 2 denotes DECREASE by 1. The agent always starts at position 0, and each move updates the position modulo 5. For example, the input **010211** corresponds to the sequence $0+1+0-1+1+1 = 2$, so the final position is 2. This task is regular because the computation can be performed by a finite-state machine with 5 states, where each state represents the current position modulo 5. Transitions are determined by interpreting the input symbol as a modular increment or decrement, making this task equivalent to simple modular arithmetic.

Reverse String (Deterministic Context-Free). Given a binary string x (e.g., **aabba**), the task is to compute its reverse y (e.g., **abbaa**). This task belongs to the class of deterministic context-free (DCF) languages because it can be solved using a pushdown automaton (PDA). The method involves pushing each input symbol onto a stack until a designated separator (such as an empty token ∅) is encountered, and then popping symbols from the stack to generate the reversed output. Since a finite-state machine lacks unbounded memory, it cannot store arbitrarily long sequences, and thus cannot solve this task. The need for a stack makes this a canonical example of a DCF problem.

Stack Manipulation (Deterministic Context-Free). Given a binary string representing the initial content of a stack (from bottom to top) and a sequence of operations—either **PUSH a**, **PUSH b**, or **POP**—the task is to execute these operations and return the final stack content from top to bottom (i.e., in the order it would be popped). For example, given the input **abbaa POP PUSH a POP**, the initial stack is **abbaa**, and applying the operations **POP**, **PUSH a**, and **POP** results in the final stack **abba**. If a **POP** operation is called on an empty stack, it is ignored. This task inherently requires the use of a stack to track and manipulate the data, making it a deterministic context-free (DCF) language task.

Bucket Sort (Context-Sensitive). Given a string over a fixed-size alphabet (e.g., size 5), the task is to return the sorted version of the string. For example, given the input **421302214**, the output is **011222344**. Since the alphabet is fixed, the task can be solved using bucket sort by maintaining one counter per symbol—requiring only a finite number of counters (5 in this case). Each counter tracks the number of occurrences of its corresponding symbol, and the final output is constructed by emitting the symbols in sorted order according to the counter values. This task is context-sensitive (CS) because it requires parallel counting and manipulation of multiple symbol classes, which goes beyond the capabilities of pushdown automata and context-free languages.

B Additional Experiment Results

We visualize the performance curves for all tasks introduced in Section 5 to enable clearer comparisons across architectures and settings. Specifically, we present two sets of plots: (1) Figure 1 shows generalization accuracy as a function of sequence length, and (2) Figure 2 shows average accuracy as a function of learning rate. These visualizations highlight how each model generalizes beyond the training length and how sensitive it is to the learning rate.

C Additional Expressivity Analysis

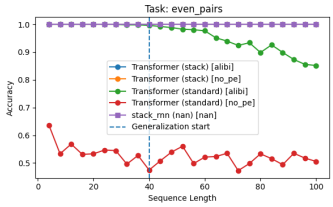
In this section, we formally demonstrate that the standard Transformer architecture *without positional encoding* cannot recognize the relatively simple regular language—Parity Checking.

Recall that in the Parity Checking task, the input is a sequence over the alphabet $\{1\}$, i.e., strings in the language $(1)^*$. The goal is to determine the number of 1s in the sequence is even or odd.

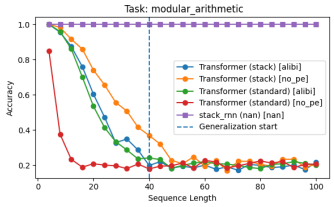
Proof. To recognize parity, a model must be able to distinguish between sequences of even and odd length. This implies that the model's output must vary depending on the input length. More precisely, the model must produce different outputs for sequences of different lengths, despite all input tokens being identical.

Let the input sequence be $w = s_1, s_2, \ldots, s_n$, where each $s_i = 1$ and $w \in (1)^*$. Because the input at every position is the same symbol, and there is no positional encoding, the input embedding at each position is the same: $h_i = h$ for all i.

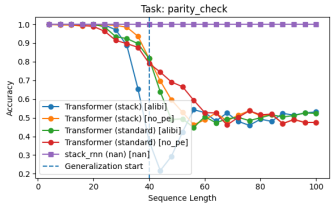
Fig. 1. Performance curves on all tasks for the Transformer encoder architecture and stack RNN, for all the positional encodings and stack augmentation, with the optimal learning rate. The dashed vertical line means the training range, meaning the sequence to the right of the dashed line have not been seen during the training and is measuring generalization.



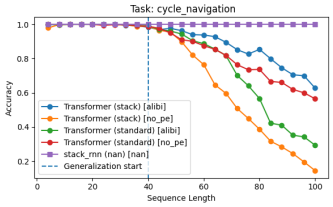
Even Pairs (R)



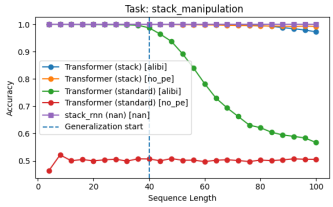
Modular Arithmetic Simple (R)



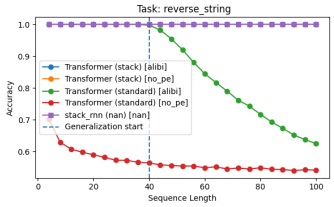
Parity Check (R)



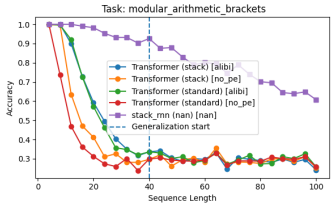
Cycle Navigation (R)



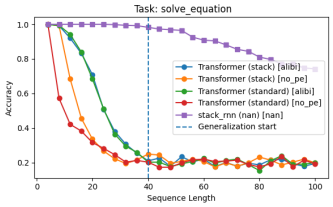
Stack Manipulation (DCF)



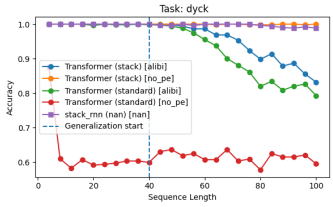
Reverse String (DCF)



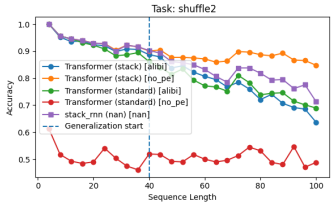
Mod. Arith Brackets (DCF)



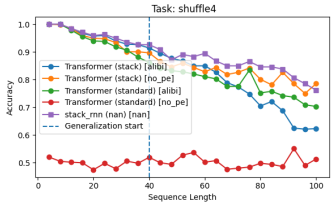
Solve Equation (DCF)



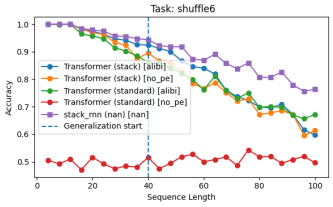
Dyck (Counter)



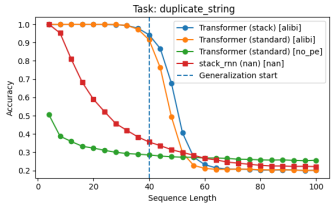
Shuffle-2 (Counter)



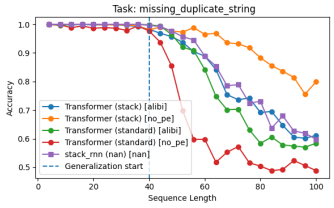
Shuffle-4 (Counter)



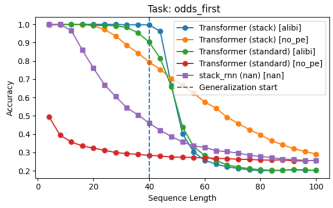
Shuffle-6 (Counter)



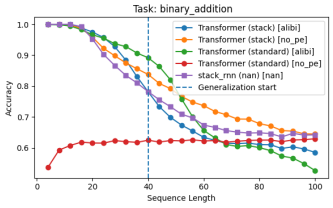
Duplicate String (CS)



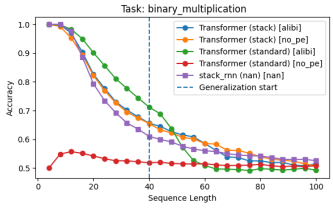
Missing Duplicate (CS)



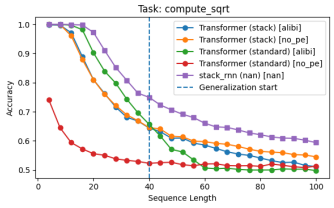
Odds First (CS)



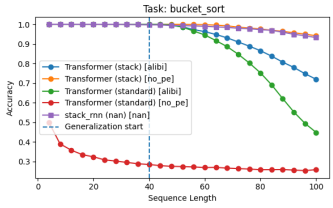
Binary Addition (CS)



Binary Multiplication (CS)



Compute Sqrt (CS)

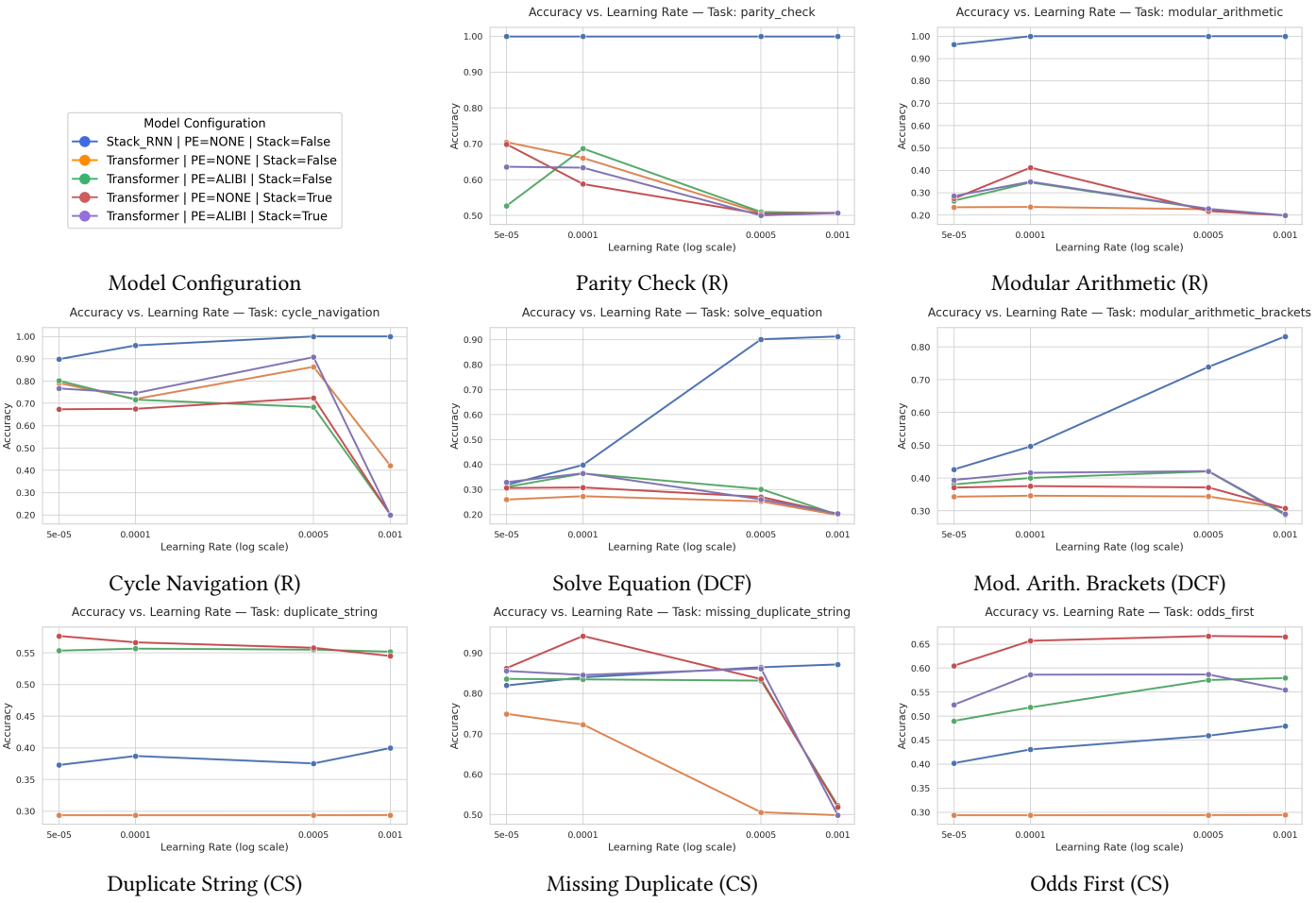


Bucket Sort (CS)

In the self-attention mechanism, the query, key, and value vectors at position i are computed as: $q_i = W_Q h, \quad k_i = W_K h, \quad v_i = W_V h,$ where W_Q, W_K, and W_V are learned matrices fixed during inference. Since all input embeddings are identical and the weight matrices are shared, all queries, keys, and values are the same across positions:

$$q_i = q, \quad k_i = k, \quad v_i = v, \quad \forall i.$$

Fig. 2. Accuracy vs. learning rate across tasks for the Transformer encoder and Stack RNN. We include all Transformer variants, presenting results both with and without positional encoding.



Model Configuration | Parity Check (R) | Modular Arithmetic (R)

Cycle Navigation (R) | Solve Equation (DCF) | Mod. Arith. Brackets (DCF)

Duplicate String (CS) | Missing Duplicate (CS) | Odds First (CS)

The attention score between position i and position j is:

$$e_{ij} = \frac{q_i^\top k_j}{\sqrt{d}} = \frac{q^\top k}{\sqrt{d}} = c,$$

where c is a constant. Therefore, the attention weights at any position i are uniform across all j:

$$\alpha_i(j) = \frac{\exp(e_{ij})}{\sum_{k=1}^{n} \exp(e_{ik})} = \frac{\exp(c)}{n\,\exp(c)} = \frac{1}{n}, \forall j$$

The output of the attention layer at position i is thus:

$$\mathrm{A}(H)_{:i} = \sum_{j=1}^{n} \alpha_i(j) v_j = \sum_{j=1}^{n} \frac{1}{n} v = v.$$

Hence, the output at each position is identical and independent of the position i or the sequence length n. Consequently, the model cannot distinguish whether the sequence has an even or odd number of tokens. Since this distinction is essential for recognizing parity, we conclude that the Transformer without positional encoding cannot recognize the Parity Checking language. □

Further, although positional encodings can mitigate the issue of identical outputs, our experiments in Section 5 show that even with positional encoding, the Transformer still generalizes poorly to longer sequences. We conjecture that this is because the positional encodings learned at the training sequence lengths do not extend effectively to lengths beyond those seen during training.

D Computational Resources And Training Time

Our experiments were conducted using four NVIDIA Tesla V100-SXM2 GPUs, each equipped with 32 GB of memory, provided by the Compute Canada high-performance computing infrastructure. We trained and evaluated our models on a total of 19 formal language tasks, spanning four regular languages, four deterministic context-free (DCF) languages, four counter languages, and seven context-sensitive (CS) languages.

For each task, we explored four different learning rates and tested the effect of positional encoding across three model configurations:

RNN, vanilla Transformer, and stack-augmented Transformer. All jobs were managed through job scripts submitted to a remote GPU cluster on Compute Canada. The cluster supports multi-GPU jobs; we parallelized each experiment into four independent sub-tasks, assigning each sub-task to one GPU to maximize efficiency and reduce training time.

To better understand the computational cost of training, we report the training time for each of the counter language tasks in Table 7. We observe that, despite its expressivity, the Stack RNN consistently requires the least amount of training time. In contrast, the Transformer model augmented with a stack-attention layer does incur a higher training cost compared to the standard Transformer. However, this increase is modest: the additional training time never exceeds 25% of the standard Transformer's runtime, and remains well within acceptable limits. We also find that, for the same Transformer architecture, enabling or disabling positional encoding has a negligible impact on training time. The difference typically remains under one minute across all experiments.

Table 7. Training times for counter (Dyck/Shuffles) language recognition tasks. Each architecture is trained with its corresponding best Hyperparameters, and the training takes 100,000 steps to complete. DY: Dyck, $S2$: Shuffle-2, $S4$: Shuffle-4, $S6$: Shuffle-6.

Task	Model	PE	Training Time
DY	Stack RNN	None	1:06:45
	Vanilla Transformer	None	1:15:30
	Vanilla Transformer	ALiBi	1:15:08
	Stack Transformer	None	1:33:55
	Stack Transformer	ALiBi	1:33:52
$S2$	Stack RNN	None	1:22:22
	Vanilla Transformer	None	1:36:20
	Vanilla Transformer	ALiBi	1:36:00
	Stack Transformer	None	1:56:02
	Stack Transformer	ALiBi	1:55:50
$S4$	Stack RNN	None	1:38:46
	Vanilla Transformer	None	1:42:24
	Vanilla Transformer	ALiBi	1:44:57
	Stack Transformer	None	1:58:14
	Stack Transformer	ALiBi	1:59:02
$S6$	Stack RNN	None	1:54:07
	Vanilla Transformer	None	1:56:52
	Vanilla Transformer	ALiBi	1:56:01
	Stack Transformer	None	2:07:31
	Stack Transformer	ALiBi	2:09:30

Received 26 July 2025; revised 26 July 2025; accepted 26 July 2025