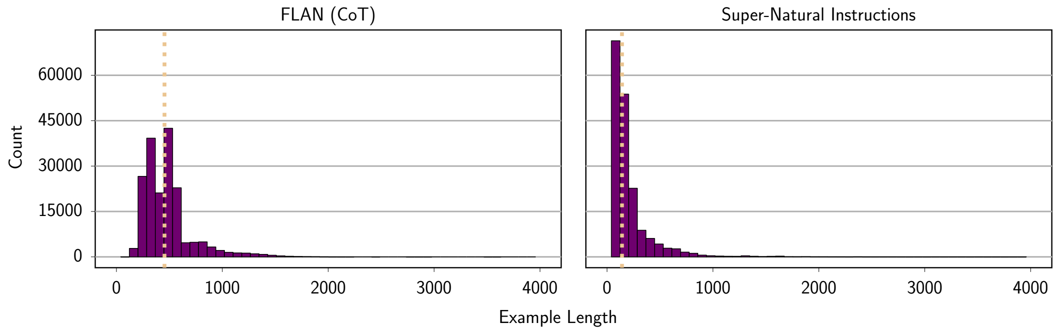

Figure A.1: Histogram of instruction lengths in two instruction finetuning datasets: FLAN (CoT subset) (Longpre et al., 2023) and Super Natural Instructions (Wang et al., 2022). The dotted line indicates the median length of the instructions in each dataset.

A Number of instances decreases rapidly as sequence length grows

The recent trend of SFT-RLHF pipeline (Ouyang et al., 2022) relies on finetuning LLMs on the instruction following tasks. However, the training data of these datasets is often skewed towards shorter sequences. Figure A.1 shows the distribution of instruction lengths in two instruction finetuning datasets: FLAN (CoT subset) (Longpre et al., 2023) and Super Natural Instructions (Wang et al., 2022). The median length of instructions in these datasets is quite short compared to the maximum length that exists. Such distribution shape highlights the importance of length generalization in these tasks. In fact, the models are supposed to learn from short instructions and generalize to ones during inference that might be much longer.

B Background

B.1 Preliminaries

In this section, we lay the groundwork and introduce the notation we use throughout the paper. We will refer to this in Appendices C.1 and C.2.

Let f_θ be a decoder-only Transformer model, where θ denotes the full set of model parameters. f_θ processes the input sequence $\boldsymbol{x} = [x_0, x_1, \dots, x_T]$ and maps it to the output sequence $\boldsymbol{y} = [y_0, y_1, \dots, y_T]$ by applying a sequence of Transformer layers. Note that being decoder-only means the attention mechanism in each layer is causal, i.e. the attention weights are computed based on the previous positions only.

The layer $\mathrm{TLayer}^{(l)}(\boldsymbol{H}^{(l-1)}; \theta_l)$, consisting of self-attention heads and a feed-forward sub-layer, reads the previous hidden state $\boldsymbol{H}^{(l-1)}$ and produces the hidden state at layer l: $\boldsymbol{H}^{l}$, where l is the layer index, and θ_l is the set of parameters of the l-th layer. Each hidden state $\boldsymbol{H}^{(l)} \in \mathbb{R}^{d \times (T+1)}$ is matrix where column t, denoted as $\boldsymbol{h}_t^{(l)}$, is the hidden state at position t.

A layer l is parameterized by a set of parameters $\theta_l = \{(\boldsymbol{W}_Q^m, \boldsymbol{W}_K^m, \boldsymbol{W}_V^m, \boldsymbol{W}_O^m)_m, \boldsymbol{W}_1, \boldsymbol{W}_2\}$, where $\boldsymbol{W}_Q^m, \boldsymbol{W}_K^m, \boldsymbol{W}_V^m \in \mathbb{R}^{h \times d}$ and $\boldsymbol{W}_O^m \in \mathbb{R}^{d \times h}$ are the query, key, value, and output matrices of the m-th head, respectively. $\boldsymbol{W}_1, \boldsymbol{W}_2 \in \mathbb{R}^{d \times k.d}$ are the weight matrices of the feed-forward sub-layer. d denotes the model's hidden state size, h is the attention dimension (where $h = \frac{d}{\text{\# heads}}$), and k is a multiplier of the hidden state size in the feed-forward sub-layer (it is usually set to 4 in common implementations of the Transformer). Note that we drop the layer index l and the attention head index m where it is clear from the context.

The Transformer layer $\mathrm{TLayer}^{(l)}$ processes each column of $\boldsymbol{H}^{(l-1)}$ independently and in parallel to produce the output. The computation of the t-th column of $\boldsymbol{H}^{(l)}$ is as follows:

$$\boldsymbol{h}_t^{(l)} = \mathrm{FF}(\lambda(\boldsymbol{a}_t + \boldsymbol{h}_t^{(l-1)})) + \boldsymbol{a}_t + \boldsymbol{h}_t^{(l-1)} \tag{3}$$

where FF is the feed-forward sub-layer, λ is layer normalization, and $\boldsymbol{a}_t \in \mathbb{R}^d$ is the output of the multi-head self-attention sub-layer at position t. Specifically, $\boldsymbol{a}_t$ is computed as:

$$\boldsymbol{a}_t = \sum_m \mathrm{Attn}^{(m)}(\boldsymbol{h}_t^{(l-1)}, \boldsymbol{H}^{(l-1)}) \tag{4}$$

where $\mathrm{Attn}^{(m)}$ is the m-th attention head. Let $\boldsymbol{o}_t \in \mathbb{R}^d$ denote the output of an attention head at position t. Then, $\boldsymbol{o}_t$ is computed as:

$$\boldsymbol{o}_t = \boldsymbol{W}_O \left(\sum_{i \leq t} \hat{\boldsymbol{\alpha}}_i \boldsymbol{v}_i \right) \tag{5}$$

where $\hat{\boldsymbol{\alpha}} = \mathrm{softmax}(\boldsymbol{\alpha}) \in \mathbb{R}^{(t+1)}$, and $\boldsymbol{\alpha}$ is the attention weight vector such that:

$$\boldsymbol{\alpha} = [\langle \boldsymbol{q}_t, \boldsymbol{k}_0 \rangle, \langle \boldsymbol{q}_t, \boldsymbol{k}_1 \rangle, \ldots, \langle \boldsymbol{q}_t, \boldsymbol{k}_t \rangle]^\intercal \tag{6}$$

where $\boldsymbol{q}_t = \boldsymbol{W}_Q \boldsymbol{h}_t^{(l-1)} \in \mathbb{R}^h$, $\boldsymbol{k}_i = \boldsymbol{W}_K \boldsymbol{h}_i^{(l-1)} \in \mathbb{R}^h$, and $\boldsymbol{v}_i = \boldsymbol{W}_V \boldsymbol{h}_i^{(l-1)} \in \mathbb{R}^h$. $\langle \cdot, \cdot \rangle$ denotes the dot product operation.

The feed-forward sub-layer $\mathrm{FF}(\cdot) \in \mathbb{R}^d$ is a two-layer MLP:

$$\mathrm{FF}(\boldsymbol{x}) = \boldsymbol{W}_2 \sigma(\boldsymbol{W}_1^\intercal \boldsymbol{x}) \tag{7}$$

where σ is a non-linear activation function (usually ReLU or GeLU (Hendrycks and Gimpel, 2020)). Additionally, $\lambda(\cdot) \in \mathbb{R}^d$ is layer normalization (Ba et al., 2016). Note that we take the *additive* (Elhage et al., 2021) view of attention heads in Equation (4) instead of *concatenate and multiple* view (Vaswani et al., 2017) as it is easier to understand and analyze. But, they are mathematically equivalent (Elhage et al., 2021).

The hidden state is initialized with a learned embedding of the input sequence $\boldsymbol{H}^{(0)} = \boldsymbol{W}_E \boldsymbol{X}$, where $\boldsymbol{W}_E \in \mathbb{R}^{d \times V}$ is the embedding matrix and $\boldsymbol{X} \in \mathbb{R}^{V \times (T+1)}$ is the one-hot encoded input sequence. V is the vocabulary size.

B.2 Positional Encoding

Almost all positional encoding methods can be explained and formulated as how they implement the dot product operation in Equation (6). So, in this section, we explain how the dot product $\langle \boldsymbol{q}_t, \boldsymbol{k}_i \rangle$ is implemented in different positional encoding schemes.

Absolute Positional Encoding (APE) The process of Absolute Positional Encoding (APE) involves assigning a position vector $\boldsymbol{p}_i$ to each absolute position i and combining them with word embeddings before inputting them into the model. So, APE first modifies how the hidden state is initialized:

$$\boldsymbol{H}^{(0)} = \boldsymbol{W}_E \boldsymbol{X} + \boldsymbol{W}_P \boldsymbol{P} \tag{8}$$

where $\boldsymbol{W}_P \in \mathbb{R}^{d \times T}$ is the positional embedding matrix and $\boldsymbol{P} \in \mathbb{R}^{V_p \times (T+1)}$ is the one-hot encoded absolute position sequence. V_p is the maximum absolute position. Therefore, the hidden state at column j is:

$$\boldsymbol{h}_j^{(0)} = \boldsymbol{e}_j + \boldsymbol{p}_j \tag{9}$$

where $\boldsymbol{e}_j \in \mathbb{R}^d$ is the word embedding of token x_j and $\boldsymbol{p}_j \in \mathbb{R}^d$ is the positional embedding for position j. Then, the dot product for the first layer in Equation (6) is computed as:

$$\begin{aligned}
\langle \boldsymbol{q}_t, \boldsymbol{k}_i \rangle &= \langle \boldsymbol{W}_Q \boldsymbol{h}_t^{(0)}, \boldsymbol{W}_K \boldsymbol{h}_i^{(0)} \rangle \\
&= \langle \boldsymbol{W}_Q (\boldsymbol{e}_t + \boldsymbol{p}_t), \boldsymbol{W}_K (\boldsymbol{e}_i + \boldsymbol{p}_i) \rangle \\
&= (\boldsymbol{W}_Q (\boldsymbol{e}_t + \boldsymbol{p}_t))^\intercal (\boldsymbol{W}_K (\boldsymbol{e}_i + \boldsymbol{p}_i)) \\
&= \boldsymbol{e}_t^\intercal \boldsymbol{W}_Q^\intercal \boldsymbol{W}_K \boldsymbol{e}_i + \boldsymbol{e}_t^\intercal \boldsymbol{W}_Q^\intercal \boldsymbol{W}_K \boldsymbol{p}_i \\
&\quad + \boldsymbol{p}_t^\intercal \boldsymbol{W}_Q^\intercal \boldsymbol{W}_K \boldsymbol{e}_i + \boldsymbol{p}_t^\intercal \boldsymbol{W}_Q^\intercal \boldsymbol{W}_K \boldsymbol{p}_i
\end{aligned} \tag{10}$$

In the learned variant of APE, $\boldsymbol{p}_j \in \mathbb{R}^d$ is learned during training. In the sinusoidal variant, $\boldsymbol{p}_j$ is calculated using a non-parametric function. Specifically, $\boldsymbol{p}_j$ is computed as:

$$\boldsymbol{p}_j = \left[\sin(\omega_1.j), \cos(\omega_1.j), \sin(\omega_2.j), \cos(\omega_2.j), \dots, \sin(\omega_{d/2}.j), \cos(\omega_{d/2}.j)\right]^\intercal \tag{11}$$

where $\omega_i = \frac{1}{10000^{2i/d}}$.

T5's Relative PE The Relative bias in T5 is a type of relative positional encoding that initially calculates the relative distance $(t-i)$ between tokens at positions t and i. This distance is then transformed into a scalar bias value b and is incorporated into the dot product between the query and key. b is learned during training. Thus, the dot product in every layer can be written as:

$$\langle \boldsymbol{q}_t, \boldsymbol{k}_i \rangle = \boldsymbol{q}_t^\intercal \boldsymbol{k}_i + b_{\text{bucket}(n-m)} \tag{12}$$

where

$$\text{bucket}(n) = \begin{cases} n & \text{if } n < \frac{\mathcal{B}}{2} \\ \frac{\mathcal{B}}{2} + \left\lfloor \frac{\log\left(\frac{n}{\mathcal{B}/2}\right)}{\log\left(\frac{\mathcal{D}}{\mathcal{B}/2}\right)} \times \frac{\mathcal{B}}{2} \right\rfloor & \text{if } \frac{\mathcal{B}}{2} \le n < \mathcal{D} \\ \mathcal{B}-1 & \text{if } n \ge \mathcal{D} \end{cases}$$

This function maps the relative distance d to a bucket index, which will be used to look up the weight corresponding to that bucket. $\mathcal{B}$ is the number of buckets, $\mathcal{D}$ is the maximum distance. It assigns half of the buckets to distances smaller than $\frac{\mathcal{D}}{2}$ with linear spacing and the other half to distances larger than $\frac{\mathcal{D}}{2}$ with logarithmic spacing. The weight for distances larger than $\mathcal{D}$ is the same. This is to facilitate generalization to unseen distances. In the original implementation of T5, $\mathcal{B} = 32$ and $\mathcal{D} = 128$. Following shows an example of the bucket function with $\mathcal{B} = 5$ and $\mathcal{D} = 6$:

$$\text{bucket}(\begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 3 & 2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 4 & 3 & 2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 5 & 4 & 3 & 2 & 1 & 0 & 0 & 0 & 0 & 0 \\ 6 & 5 & 4 & 3 & 2 & 1 & 0 & 0 & 0 & 0 \\ 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 & 0 & 0 \\ 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 & 0 \\ 9 & 8 & 7 & 6 & 5 & 4 & 3 & 2 & 1 & 0 \end{bmatrix}) = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 3 & 2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 3 & 3 & 2 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 4 & 3 & 3 & 2 & 1 & 0 & 0 & 0 & 0 & 0 \\ 4 & 4 & 3 & 3 & 2 & 1 & 0 & 0 & 0 & 0 \\ 4 & 4 & 4 & 3 & 3 & 2 & 1 & 0 & 0 & 0 \\ 4 & 4 & 4 & 4 & 3 & 3 & 2 & 1 & 0 & 0 \\ 4 & 4 & 4 & 4 & 4 & 3 & 3 & 2 & 1 & 0 \end{bmatrix}$$

ALiBi Similar to T5's Relative PE, ALiBi subtracts a scalar bias from the attention score. As the distance between the query and key tokens increases, the bias grows linearly. Specifically, the dot product in every layer can be written as:

$$\langle \boldsymbol{q}_t, \boldsymbol{k}_i \rangle = \boldsymbol{q}_t^\intercal \boldsymbol{k}_i - (t-i).C^{(m+1)} \tag{13}$$

where m is head index and C is a constant defined as:

$$C = 2^{-2^{-\log_2(\#\text{heads}+3)}}$$

For example, if the number of heads is 8, then we have $\frac{1}{2}, \frac{1}{2^2}, \dots, \frac{1}{2^8}$ (Press et al., 2022).

Rotary The Rotary is a relative PE that applies a rotation to the query and key representations based on their absolute positions before dot product attention. Due to this rotation, the attention dot product relies solely on the relative distance between tokens.

First, we formulate Rotary for model dimension $d = 2$. Rotary positional encoding defines the dot product as:

$$
\begin{aligned}
\langle \boldsymbol{q}_t, \boldsymbol{k}_i \rangle =& \langle \mathrm{Rot}(\boldsymbol{q}_t, t), \mathrm{Rot}(\boldsymbol{k}_i, i) \rangle \\
=& \langle \boldsymbol{R}^{t\theta} \boldsymbol{q}_t, \boldsymbol{R}^{i\theta} \boldsymbol{k}_i \rangle \\
=& (\boldsymbol{R}^{t\theta} \boldsymbol{q}_t)^\intercal (\boldsymbol{R}^{i\theta} \boldsymbol{k}_i) \\
=& \boldsymbol{q}_t{}^\intercal (\boldsymbol{R}^{t\theta})^\intercal \boldsymbol{R}^{i\theta} \boldsymbol{k}_i \\
=& \boldsymbol{q}_t{}^\intercal \boldsymbol{R}^{(i-t)\theta} \boldsymbol{k}_i
\end{aligned}
\tag{14}
$$

where $\boldsymbol{R}^{t\theta}$ is a rotation matrix that rotates $\boldsymbol{x}$ by $t\theta$ radians:

$$
\boldsymbol{R}^{n\theta} = \begin{bmatrix} \cos(n\theta) & -\sin(n\theta) \\ \sin(n\theta) & \cos(n\theta) \end{bmatrix} \tag{15}
$$

for $d > 2$, Rotary applies the same approach on every two consecutive dimensions of $\boldsymbol{q}_t$ and $\boldsymbol{k}_i$, but with different θ angles. Refer to Su et al. (2021) for the exact formulation.

NoPE NoPE does not explicitly encode positional encodings. So, the dot product in every layer can be written as:

$$
\langle \boldsymbol{q}_t, \boldsymbol{k}_i \rangle = \boldsymbol{q}_t^\intercal \boldsymbol{k}_i \tag{16}
$$

C Proofs

In this section, we provide proof of why NoPE can implicitly learn both absolute and relative positions. We refer the readers to Appendix B.1 for the notation and definitions used in this section.

C.1 Absolute Positional Encoding in NoPE

This section discusses how NoPE can recover absolute positions in the hidden state. Our proof is inspired by Weiss et al. (2021); Lindner et al. (2023) and relies on the causal attention mask in the decoder-only Transformer and the $\mathrm{softmax}$ function to recover absolute positions.

Theorem 1 (Absolute Encoding). *Let $\boldsymbol{x} = [\texttt{<bos>}, x_1, \ldots, x_T]$ be an input sequence of length $T+1$ to the model. Then, the first layer of f_θ can recover absolute positions $[1, \ldots, T+1]$ in the hidden state $\boldsymbol{H}^{(1)}$. That is, there exist $\boldsymbol{W}_Q$, $\boldsymbol{W}_K$, $\boldsymbol{W}_V$, $\boldsymbol{W}_O$, $\boldsymbol{W}_1$, and $\boldsymbol{W}_2$ such that the self-attention and feedforward operations in the first layer compute absolute positions and write it to the next hidden state.*

Proof.

Our proof only specifies the weights of a single attention head in the first layer (and additionally the parameterization of feedforward sub-layer). In this parameterization, we only require the first three dimensions of the hidden states. The rest of the heads, as long as they do not override the first three dimensions, can be arbitrary. This does not impose any challenges as Transformers used in practice usually have a very large model dimension d. In the rest, we provide the construction of the weights and then verify that they can recover absolute positions.

First, we construct the word embedding matrix $\boldsymbol{W}_E \in \mathbb{R}^{d \times V}$, where each column is the embedding of a token in the vocabulary. We construct $\boldsymbol{W}_E$ such that it always sets the first dimension of every embedding vector to be 1. Additionally, it sets the second dimension to 1 if and only if the token is <bos>. Otherwise, it sets it to zero. The third dimension of all embedding vectors is set to zero. Other dimensions can take any arbitrary values. Without loss of generality, assume <bos> is the first token in the vocabulary, i.e. The first column. Then, we have:

$$
\boldsymbol{W}_E = \begin{bmatrix}
1 & 1 & 1 & \ldots & 1 \\
1 & 0 & 0 & \ldots & 0 \\
0 & 0 & 0 & \ldots & 0 \\
e_{4,1} & e_{4,2} & e_{4,3} & \ldots & e_{4,V} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
e_{d,1} & e_{d,2} & e_{d,2} & \ldots & e_{d,V}
\end{bmatrix}_{d \times V} \tag{17}
$$

where $e_{d,i} \in \mathbb{R}$.

Secondly, for head dimensions $h \geq 1$, we construct the weights $\boldsymbol{W}_Q, \boldsymbol{W}_K, \boldsymbol{W}_V, \boldsymbol{W}_O$ of the first attention head in the first layer. Specifically,

$$\boldsymbol{W}_K = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 1 & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 0 & \dots & 0 \end{bmatrix}_{h \times d} \quad \boldsymbol{W}_V = \begin{bmatrix} 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & 0 \end{bmatrix}_{h \times d} \tag{18}$$

$\boldsymbol{W}_K$ reads from the first dimension of the hidden state, which is initialized with 1 using the embedding matrix. Since all word embeddings have one in their first dimension, this parameterization will result all key vectors to be the same. Moreover, $\boldsymbol{W}_V$ reads from the second dimension of the hidden state, which is initialized with 1 if the token is `<bos>`. So, the value vector will have 1 in its first dimension only if the corresponding token is `<bos>`.

$\boldsymbol{W}_Q$ can be any arbitrary matrix. $\boldsymbol{W}_O$ will write the result of the attention to the third dimension of the hidden state and can be constructed as:

$$\boldsymbol{W}_O = \begin{bmatrix} 0 & 0 & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & 0 & \dots & 0 \\ 1 & 0 & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \dots & 0 \end{bmatrix}_{d \times h} \tag{19}$$

Now, we verify that for any input sequence $\boldsymbol{x} = [\texttt{<bos>}, x_1, \dots, x_T]$, the first layer can recover absolute positions $[1, \dots, T+1]$ in the hidden state $\boldsymbol{H}^{(1)}$. We verify this for column t of $\boldsymbol{H}^{(1)}$. That is, we show that absolute position information is available in the third dimension of $\boldsymbol{h}_t^{(1)}$.

First, we use the word embedding matrix $\boldsymbol{W}_E$ to compute the embedding $\boldsymbol{H}^{(0)}$:

$$\boldsymbol{H}^{(0)} = \boldsymbol{W}_E \boldsymbol{X} = \begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & 0 & 0 & \dots & 0 \\ 0 & 0 & 0 & \dots & 0 \\ e_{4,1} & e_{4,2} & e_{4,3} & \dots & e_{4,V} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ e_{d,1} & e_{d,2} & e_{d,2} & \dots & e_{d,V} \end{bmatrix}_{d \times (T+1)} \tag{20}$$

We now provide the attention computation at position $1 \leq t \leq T+1$. First, we use $\boldsymbol{W}_Q$ to compute the query vector $\boldsymbol{q}_t$ by applying $\boldsymbol{q}_t = \boldsymbol{W}_Q \boldsymbol{h}_t^{(0)}$:

$$\boldsymbol{q}_t = [q_1, q_2, q_3, \dots, q_h]^\intercal \tag{21}$$

Recall that $\boldsymbol{W}_Q$ can be any arbitrary matrix. So, $q_j \in \mathbb{R}$ can take any arbitrary value. Next, we compute the key vectors by applying $\boldsymbol{k}_i = \boldsymbol{W}_K \boldsymbol{h}_i^{(0)}$:

$$\boldsymbol{k}_1 = \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix} \quad \boldsymbol{k}_2 = \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix} \quad \dots \quad \boldsymbol{k}_t = \begin{pmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix} \tag{22}$$

Note that all key vectors are the same and we only need to compute them up to position t as the attention mask is causal, i.e query can only look at positions $\leq t$. Next, we compute the attention weight vectors $\boldsymbol{\alpha}$:

$$\boldsymbol{\alpha} = [\langle \boldsymbol{q}_t, \boldsymbol{k}_1 \rangle, \langle \boldsymbol{q}_t, \boldsymbol{k}_2 \rangle, \dots, \langle \boldsymbol{q}_t, \boldsymbol{k}_t \rangle]^\intercal \tag{23}$$
$$= [\alpha^*, \alpha^*, \dots, \alpha^*]^\intercal \tag{24}$$

where $\alpha^* = q_1 + q_2 + \ldots + q_h$. Next, we apply softmax to compute the attention probabilities. Since all $\boldsymbol{\alpha}^i$'s are the same, we have:

$$\hat{\boldsymbol{\alpha}} = \mathrm{softmax}(\boldsymbol{\alpha}) = \left[\frac{1}{t}, \frac{1}{t}, \ldots, \frac{1}{t}\right]^\intercal \tag{25}$$

Now, we compute the value vectors by applying $\boldsymbol{v}_i = \boldsymbol{W}_V \boldsymbol{h}_i^{(0)}$:

$$\boldsymbol{v}_1 = \begin{pmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \quad \boldsymbol{v}_2 = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \quad \ldots \quad \boldsymbol{v}_t = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix} \tag{26}$$

Finally, we compute the output of the attention head by applying $\boldsymbol{W}_O$:

$$\boldsymbol{o}_t = \boldsymbol{W}_O \left(\sum_{i \le t} \hat{\boldsymbol{\alpha}}_i \boldsymbol{v}_i \right) = \boldsymbol{W}_O \left(\frac{1}{t} \sum_{i \le t} \boldsymbol{v}_i \right) = \boldsymbol{W}_O \begin{pmatrix} 1/t \\ 0 \\ \vdots \\ 0 \end{pmatrix}_h = \begin{pmatrix} 0 \\ 0 \\ 1/t \\ 0 \\ \vdots \\ 0 \end{pmatrix}_d \tag{27}$$

Thus, the output of our constructed attention head recovers the absolute position information and writes it to the third dimension of output.

We used the decoder-only property of Transformer implicitly in Equation (23), which helped us to only attend to position $\le t$. So, the lengths of the attended sequence are always t. Moreover, the presence of <bos> token in the input sequence helped us to anchor the absolute position information. This is not a problem as in practice models are often prompted with some instructions which can act as <bos> token.

With this information available to the rest of the network, the feedforward sub-layer, with sufficient hidden width, can recover the absolute positions $[1, 2, \ldots, T+1]$ from the third dimension of attention output. This is because the feedforward sub-layer is MLP with ReLU activation. So, it can learn any arbitrary function (Park et al., 2020). Note that the layer-norm operation can be bypassed as explained by Akyurek et al. (2023). □

C.2 Relative Positional Encoding in NoPE

In this section, we show if the hidden state contains absolute positional information as explained in the previous section, then the attention mechanism in all subsequent layers can implement a relative positional encoding. We refer the readers to Appendices B.1 and C.1 for the notation and definitions used in this section.

Theorem 2 (Relative Encoding). *Suppose that the hidden state $\boldsymbol{H}^{(1)}$ contains absolute positional information, as stated in Theorem 1, and assume that it is not overwritten by any subsequent layers. Then, the self-attention in all subsequent layers can implement a relative positional encoding: there exists a parameterization of f_θ such that, for $l \ge 2$, the attention dot product between query $\boldsymbol{q}_t$ and key $\boldsymbol{k}_i$ at positions t and i ($t \ge i$) can be expressed as:*

$$\langle \boldsymbol{q}_t, \boldsymbol{k}_i \rangle = f_{\mathrm{cnt}}(\boldsymbol{q}_t, \boldsymbol{k}_i) + f_{\mathrm{rel}}(t - i) \tag{1}$$

where f_{cnt} is a function of their content, and f_{rel} is a function of their relative distance.

Proof.

Our proof only specifies a few entries of weight matrices for attention heads in layers $l \ge 2$, which does not impose any challenges for Transformers used in practice as they usually have a very large model dimension d. Moreover, we require to have absolute positions in the third dimension of the hidden state as explained in Theorem 1. To show NoPE can implement relative encoding, we only need to prove that its attention dot product depends on the relative distance between tokens (See

Appendix B.1 for an overview of relative encoding methods). In the rest, we provide the construction of the weights and then verify that they can implement relative position encoding.

For head dimension $h \geq 2$, we construct the weights $\boldsymbol{W}_Q, \boldsymbol{W}_K$ of the attention heads in the second layers and above. Specifically,

$$\boldsymbol{W}_Q = \begin{bmatrix} 1 & 0 & 0 & 0 & \dots & 0 \\ 0 & 0 & -1 & 0 & \dots & 0 \\ w_{3,1} & w_{3,2} & w_{3,3} & w_{3,4} & \dots & w_{3,d} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ w_{h,1} & w_{h,2} & w_{h,3} & w_{h,4} & \dots & w_{h,d} \end{bmatrix}_{h \times d} \tag{28}$$

$$\boldsymbol{W}_V = \begin{bmatrix} 0 & 0 & 1 & 0 & \dots & 0 \\ 1 & 0 & 0 & 0 & \dots & 0 \\ w'_{3,1} & w'_{3,2} & w'_{3,3} & w'_{3,4} & \dots & w'_{3,d} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ w'_{h,1} & w'_{h,2} & w'_{h,3} & w'_{h,4} & \dots & w'_{h,d} \end{bmatrix}_{h \times d} \tag{29}$$

where $w_{i,j}, w'_{i,j} \in \mathbb{R}$ can take any arbitrary value. Their corresponding $\boldsymbol{W}_V$ and $\boldsymbol{W}_O$ can take any arbitrary values as long as they do not override the first three dimensions of the residual stream.

Now we verify that for any input sequence $\boldsymbol{x} = [\texttt{<bos>}, x_1, \ldots, x_T]$, the attention dot product between query $\boldsymbol{q}_t$ and key $\boldsymbol{k}_i$ at positions t and i ($t \geq i$) will depend the relative distance between tokens.

First, assume that absolute positions are computed in the hidden state $\boldsymbol{H}^{(l)}$ for $l \geq 1$, as stated in Theorem 1. Specifically,

$$\boldsymbol{H}^{(l)} = \begin{bmatrix} 1 & 1 & 1 & 1 & \dots & 1 \\ 1 & 0 & 0 & 0 & \dots & 0 \\ 1 & 2 & 3 & 4 & \dots & T+1 \\ h_{4,1} & h_{4,2} & h_{4,3} & h_{4,4} & \dots & h_{4,T+1} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ h_{d,1} & h_{d,2} & h_{d,3} & h_{d,4} & \dots & h_{d,T+1} \end{bmatrix}_{d \times (T+1)} \tag{30}$$

where $h_{i,j} \in \mathbb{R}$ can be any arbitrary value as the first three dimensions of the hidden state are reserved for PE computation. The rest of the dimensions can take any arbitrary values as in regular computation of Transformers.

We now present the attention computations at position $1 \leq t \leq T+1$. We use $\boldsymbol{W}_Q$ to compute the query vector $\boldsymbol{q}_t$ by applying $\boldsymbol{q}_t = \boldsymbol{W}_Q \boldsymbol{h}_t^{(l)}$:

$$\boldsymbol{q}_t = [1, -t, q_3, \ldots, q_h]^\intercal \tag{31}$$

where $q_j \in \mathbb{R}$ can take any arbitrary value. Next, we compute the key vectors by applying $\boldsymbol{k}_i = \boldsymbol{W}_K \boldsymbol{h}_i^{(l)}$:

$$\boldsymbol{k}_1 = \begin{pmatrix} 1 \\ 1 \\ k_{3,1} \\ \vdots \\ k_{h,1} \end{pmatrix} \quad \boldsymbol{k}_2 = \begin{pmatrix} 2 \\ 1 \\ k_{3,2} \\ \vdots \\ k_{h,2} \end{pmatrix} \quad \boldsymbol{k}_3 = \begin{pmatrix} 3 \\ 1 \\ k_{3,3} \\ \vdots \\ k_{h,3} \end{pmatrix} \quad \dots \quad \boldsymbol{k}_t = \begin{pmatrix} t \\ 1 \\ k_{3,t} \\ \vdots \\ k_{h,t} \end{pmatrix} \tag{32}$$

where $k_{(\cdot,\cdot)} \in \mathbb{R}$ can have any arbitrary value. So, for $\boldsymbol{k}_i$ we have:

$$\boldsymbol{k}_i = [i, 1, k_{3,i}, \ldots, k_{h,i}]^\intercal \tag{33}$$

Next, we let us present the attention dot product between $\boldsymbol{q}_t$ and $\boldsymbol{k}_i$:

$$\begin{aligned}
\langle \boldsymbol{q}_t, \boldsymbol{k}_i \rangle =& 1.i + (-t).1 + q_3.k_{3,i} + \cdots + q_h.k_{h,i} && (34)\\
=& i - t + \sum_{j=3}^{h} q_j.k_{j,i} && (35)\\
=& \left(\sum_{j=3}^{h} q_j.k_{j,i} \right) - (t - i) && (36)\\
=& f_{\text{cnt}}(\boldsymbol{q}_t, \boldsymbol{k}_i) + f_{\text{rel}}(t - i) && (37)
\end{aligned}$$

Thus, the dot product between $\boldsymbol{q}_t$ and $\boldsymbol{k}_i$ depends on the relative distance between tokens (assuming the rest of the terms do not cancel out which can be easily avoided by setting the respective weights in Equations (28) and (29)). Note that our proof uses the linear spacing between tokens, but the MLP the first layer can write any arbitrary function of absolute positions to the third dimension of the hidden state, which enables more complex relative encoding schemes. □

D Experimental Details

D.1 Tasks

Here we provide the details and more examples of the tasks and datasets we used in our evaluation. For each task, we sample 100K examples for the training set and 10K for the test. Also, we use 15% of the train as the validation set.

Addition The addition task (Nye et al., 2021) asks the model to compute the sum of two numbers. Each number is represented as a sequence of digits that are separated by space. So, the model has access to the exact digits.

Input

```
Compute:  5 3 7 2 6 + 1 9 1 7 ?
```

Output

```
The answer is 5 5 6 4 3.
```

we create each length bucket based on the number of digits in each number, e.g. 6-by-3, 6-by-4, etc. For the training set, we use buckets where one of the numbers has at most L digits. For the test set, we use buckets where any of the numbers have at most L digits. The model is evaluated on the correctness of its predicted result.

Polynomial Evaluation The polynomial evaluation task (Nye et al., 2021) asks the model to evaluate a polynomial expression at a given value of x. The polynomial terms and digits are separated to make just the tokenizer does not glue symbols together.

Input

```
Evaluate x = 3 in ( 3 x ** 0 + 1 x ** 1 + 1 x ** 2 ) % 10 ?
```

Output

```
The answer is 5.
```

The length bucket is created based on the number of terms in the polynomial expression. We sample x from $\mathcal{U}(-2, 2)$, the degree of each term from $\mathcal{U}(0, 3)$, and the coefficient of each term from $\mathcal{U}(-3, 3)$. We take the modulo of the result by 10 to make the task easier for the model and make sure we only measure the generalization of the length of the problem instance not the value of the polynomial. The model is evaluated on the correctness of its predicted result.

Sorting The sorting task (Saxton et al., 2019) asks the model to sort a sequence of input numbers. We use this task in two variants: Single Token and Multi Digit. In the Single Token variant, we create an alphabet of 50 tokens from the model's vocabulary and fix some canonical ordering among them through task. Each instance is a sequence of tokens from the alphabet in a random order, and the model is asked to sort them in the canonical order.

Input

```
Sort the following numbers:  3 1 4 1 5 ?
```

Output

```
The answer is 1 1 3 4 5.
```

In the Multi Digit variant, we simply present a sequence of multi digit/tokens numbers to the model, and ask it to sort them in ascending order. Each number is represented by its digits and they are separated by a space.

Input

```
Sort the following numbers:  3 1, 4 1, 5 9, 1 2 6, 5 3 3 ?
```

Output

```
The answer is 3 1, 4 1, 5 9, 1 2 6, 5 3 3.
```

In this case, we sample each number from $\mathcal{U}(0, 10000)$. In both cases, the length bucket is created based on the length of the input sequence. The model is evaluated on the correctness of its predicted result.

Summation In this task (Saxton et al., 2019), we ask the model to compute the sum of a sequence of input numbers modulo 10 as we want to specifically measure how the model generalizes to longer sequences not the value of summation result:

Input

```
Compute:  ( 1 + 2 + 3 + 4 + 7 ) % 10 ?
```

Output

```
The answer is 7.
```

Each digit is randomly sampled from $\mathcal{U}(1, 9)$. The length bucket is created based on the length of the input sequence. The model is evaluated on the correctness of its predicted result.

Parity In the parity task (Anil et al., 2022), we ask the model to compute the parity of a binary sequence.

Input

```
Is the number of 1's even in [ 1 0 0 1 1] ?
```

Output

```
The answer is No.
```

LEGO In the LEGO task (Zhang et al., 2023), the model is provided with a simple computation graph (DAG), where each node represents a variable, and variables are connected by simple operations which created the edges in the computation graph. We refer to Zhang et al. (2023) for a detailed description.

Input

```
If a = -1; b = -a; c = +b; d = +c.  Then what is c?
```

Output

```
The answer is +1.
```

To sample each example, we first sample the list of variables based on the length of the example, and then we uniformly sample the value of each variable to make sure all variables are represented with both -1 and +1. Finally, given the value of variables, we deterministically compute the operation on each edge. For each example, we query all variables from the middle of the computation graph to the end. The model is evaluated on the correctness of its predicted result.

Copy The copy task is straightforward. The model has to repeat the input sequence in the output.

Input

```
Copy the following words: <w1> <w2> <w3> <w4> <w5> .
```

Output

```
<w1> <w2> <w3> <w4> <w5>
```

We create multiple variants of this task to better understand the models' generalization behavior. In the first variant, the input tokens are the same, so the model has to basically count the number of input tokens. In the second variant, the model has to replace the input tokens with another token sampled from the vocabulary. In the third variant, we sample the input tokens from the model's vocabulary, and the model has to predict them in the same order. We also create 2x versions of variants 1 and 3 to make the tasks more challenging.

Reverse In this task the model, the model has to reverse the order of input tokens in its output.

Input

```
Reverse the following words: <w1> <w2> <w3> <w4> <w5> .
```

Output

```
<w5> <w4> <w3> <w2> <w1> .
```

As in the copy task, we create multiple variants of this task. In the first variant, the model has to reverse the order of input tokens, where the tokens are randomly sampled from the model's vocabulary. In the second variant, the model has to reverse the order of input tokens, as in the first variant, but also it has to reverse it one more time, recreating the original input.

D.2 Hyperparameters

Table 2 shows the hyperparameters we used in our experiments. We use the same hyperparameters for all models and positional encoding schemes. In our initial experiment, we tried a few more hyperparameters such as $\text{lr} \in \{0.00001, 0.00003, 0.00005\}$ and $\text{WeightDecay} \in \{0, 0.05, 0.1\}$, but we did not observe any significant difference in the results. So, we decided to use the same hyperparameters throughout our experiments.

D.3 Compute

In our experiments, we used single-GPU training setup for the models. Specifically, we ran our experiments on a mix of NVIDIA V100 32G, NVIDIA RTX8000 48G, NVIDIA A100 40G, and NVIDIA A100 80G GPUs. Depending on the GPU type and the positional encoding, each of our training runs took 6 to 15 hours, per each seed, on average to complete. Considering all the datasets, and positional encoding schemes, in addition to the scratchpad experiments, and three seeds, we ran about 870 individual training runs for results in this paper.

D.4 Reproducibility

In this study, all experiments employed open-source libraries, specifically HuggingFace (Wolf et al., 2020) from which we utilized their implementation as a foundation for the training loop, optimizer, and the Transformer architecture. To ensure the reproducibility, we will also release a singularity binary with all dependencies and libraries to enable running our experiments on any machine with NVIDIA GPUs and at anytime in the future. Moreover, every reported number in this paper is linked to source code package that deterministically (up to GPU stochasticity) reproduces the results, which we release them publicly on GitHub at `https://www.omitted.link`.

E Full Results

E.1 Detailed Model Accuracy

We report the detailed results of our experiments in Figures E.2 to E.4. We refer the readers to Appendix D.1 for the description of each task.

Table 2: Summary of hyperparamters used in the experiments.

Parameter	Value
Optimizer	AdamW
Learning rate	0.00003
Weight Decay	0.05
Batch size	64
Learning Rate Scheduler	Polynomial
Warm Up	6% of training steps
# Train Steps	40K steps
Dropout *(taken from HuggingFace)*	0.1
Model dimension *(taken from HuggingFace)*	768
# Layers *(taken from HuggingFace)*	12
# Attention Heads *(taken from HuggingFace)*	12

E.2 Detailed Head Distance

Figure E.5 shows the layer-wise distance of No PE's attention patterns with other positional encoding schemes measured across instances of the SCAN dataset. We refer the readers to Section 5.2 for the details and analysis of these results.

E.3 Detailed Model Accuracy On Various Scratchpad Formats

Figure E.6 shows the generalization of various scratchpad formats for each model aggregated across all datasets. Figures E.7 to E.13 show the generalization of various scratchpad formats for each model on each dataset. We refer the readers to Section 6 for the details and analysis of these results.

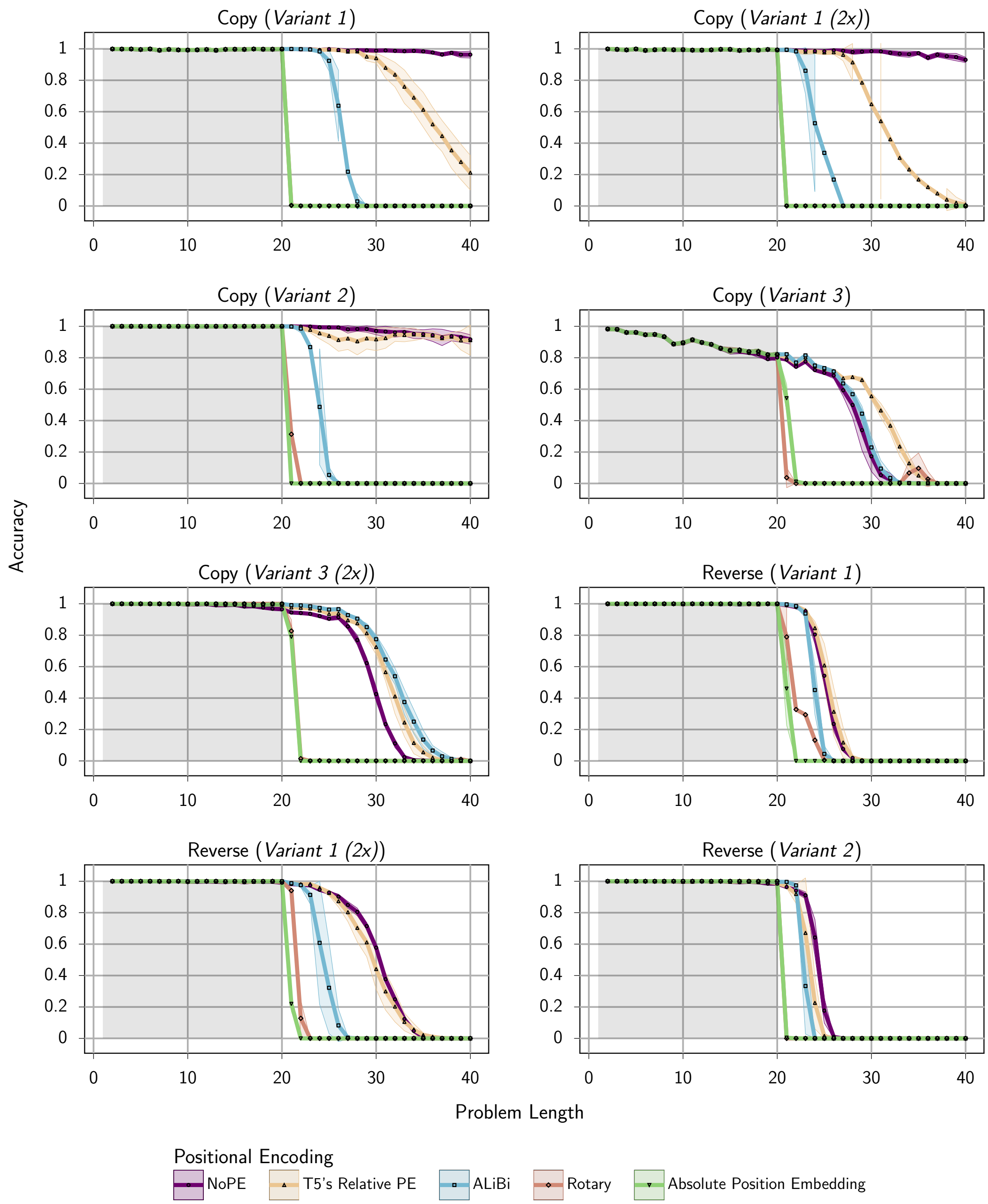

Figure E.2: Generalization behavior of positional encoding schemes on Primitive tasks.

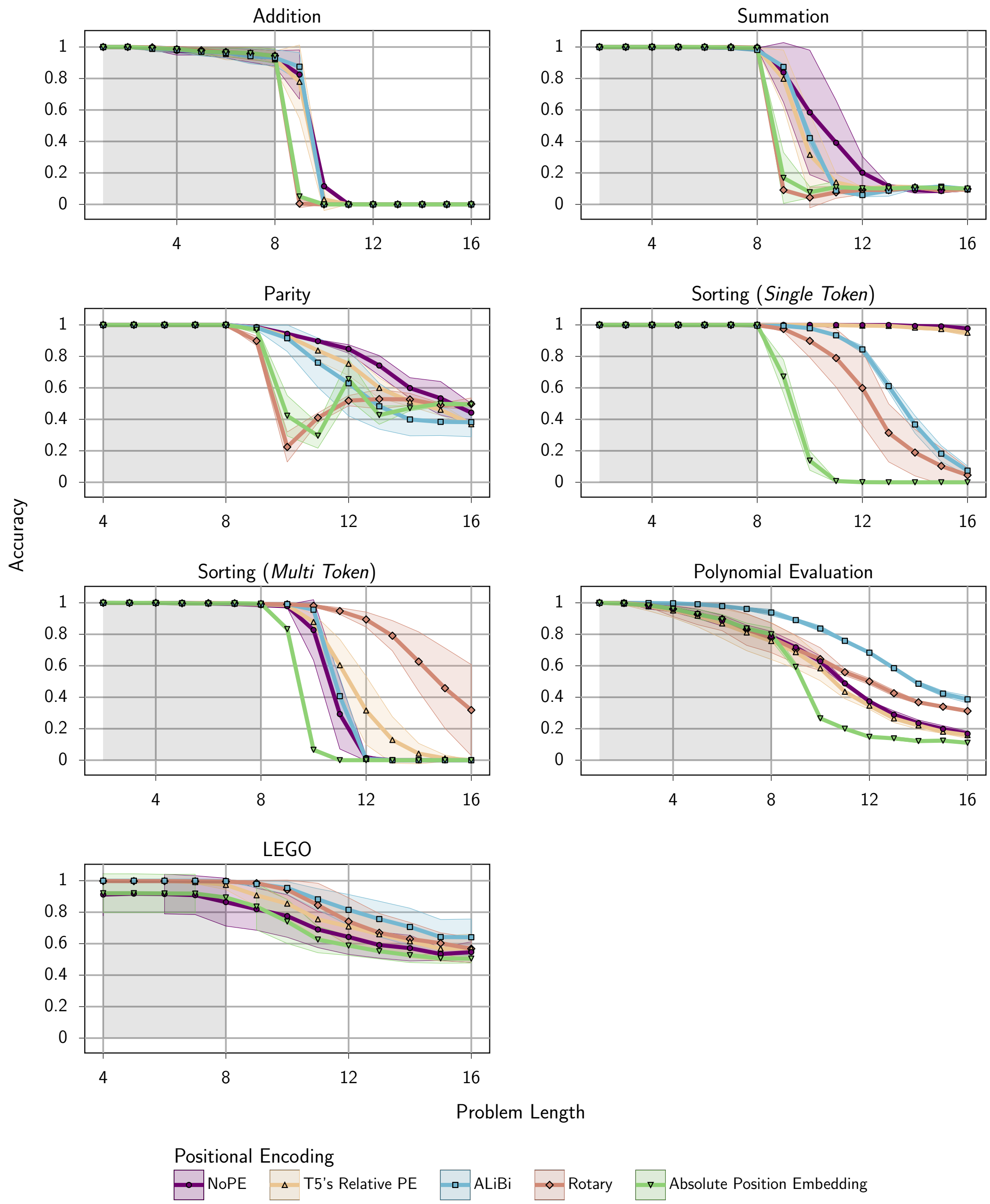

Figure E.3: Generalization behavior of positional encoding schemes on Mathematical & Reasoning tasks.

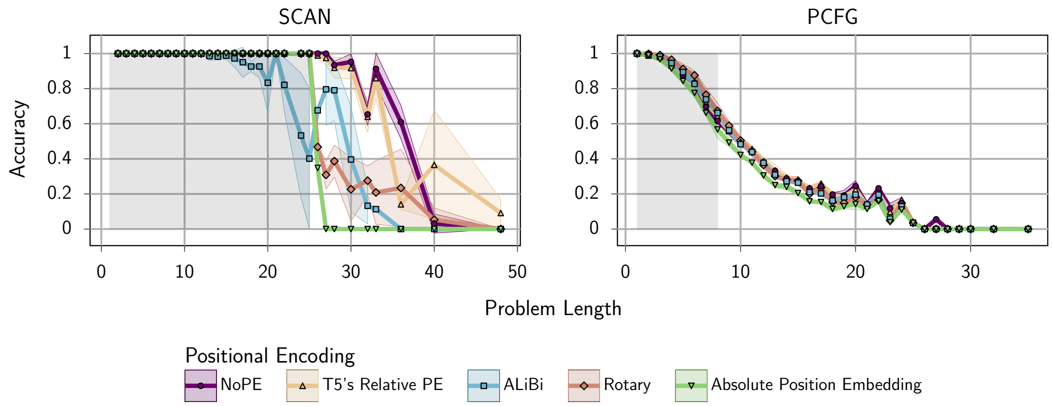

Figure E.4: Generalization behavior of positional encoding schemes on Classic Length Generalization tasks.

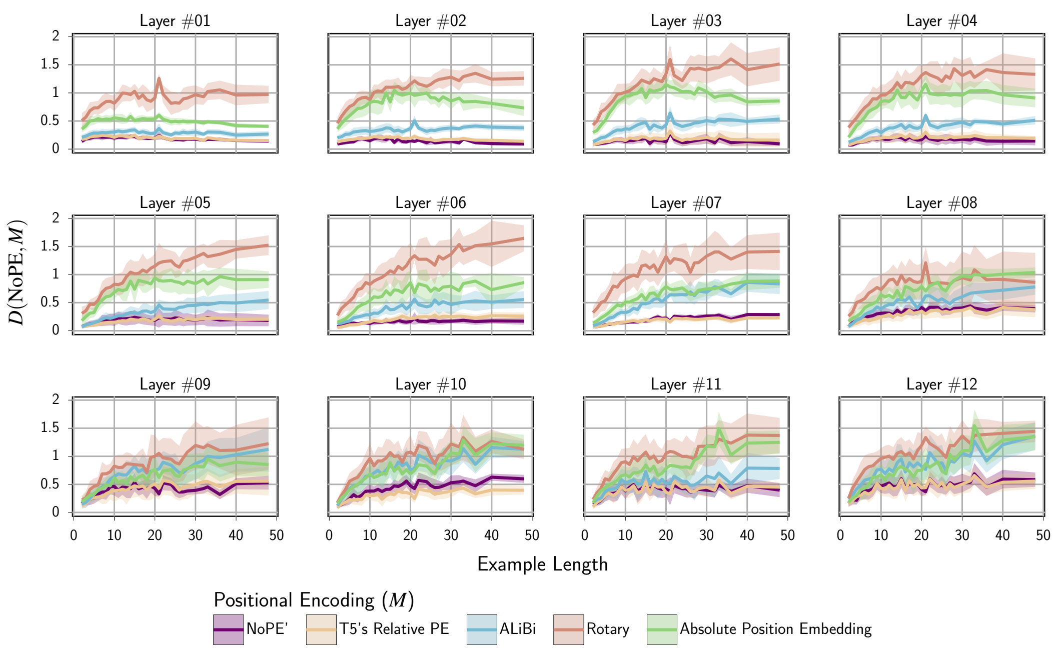

Figure E.5: Layer-wise distance of No PE's attention patterns with other positional encoding schemes measured across instances of SCAN dataset.

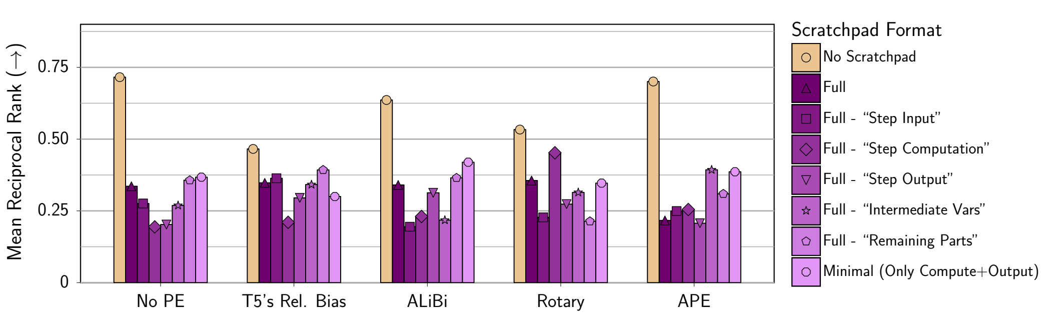

Figure E.6: The optimal scratchpad format is aggregated across all datasets per each model. The optimal scratchpad format is different for each model.

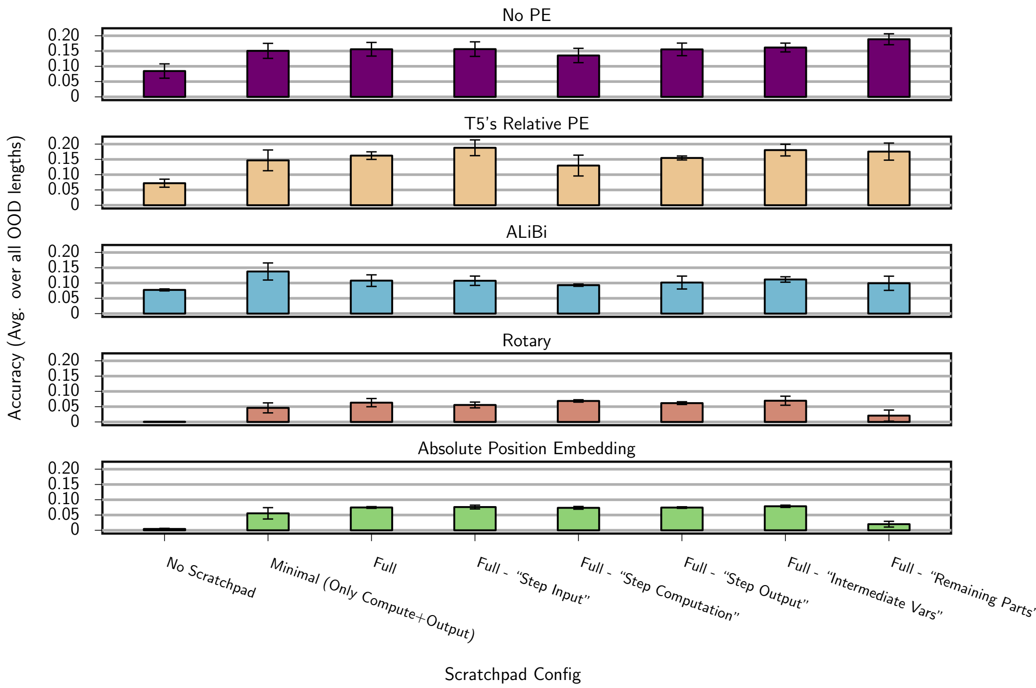

Figure E.7: Generalization of various scratchpad formats for each model on the **Addition** task.

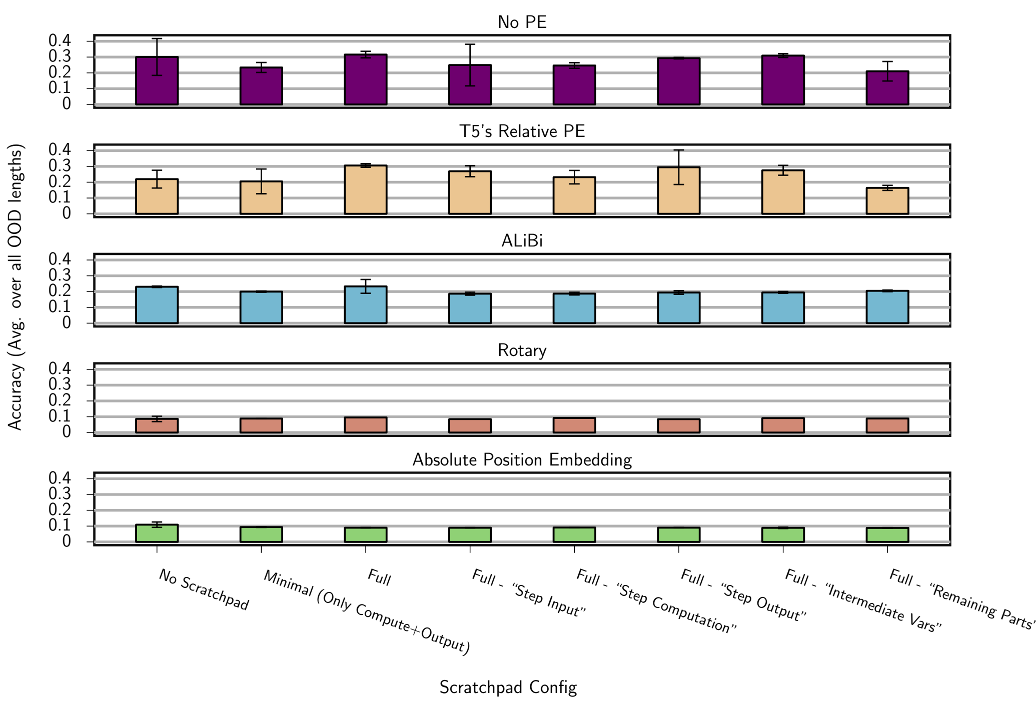

Figure E.8: Generalization of various scratchpad formats for each model on the **Summation** task.

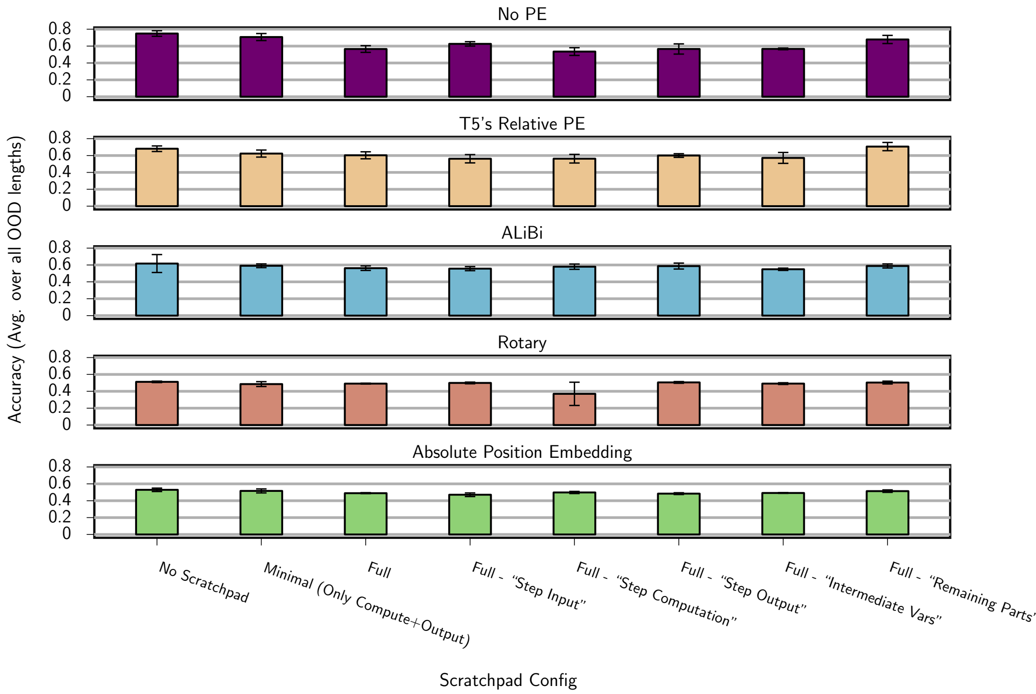

Figure E.9: Generalization of various scratchpad formats for each model on the **Parity** task.

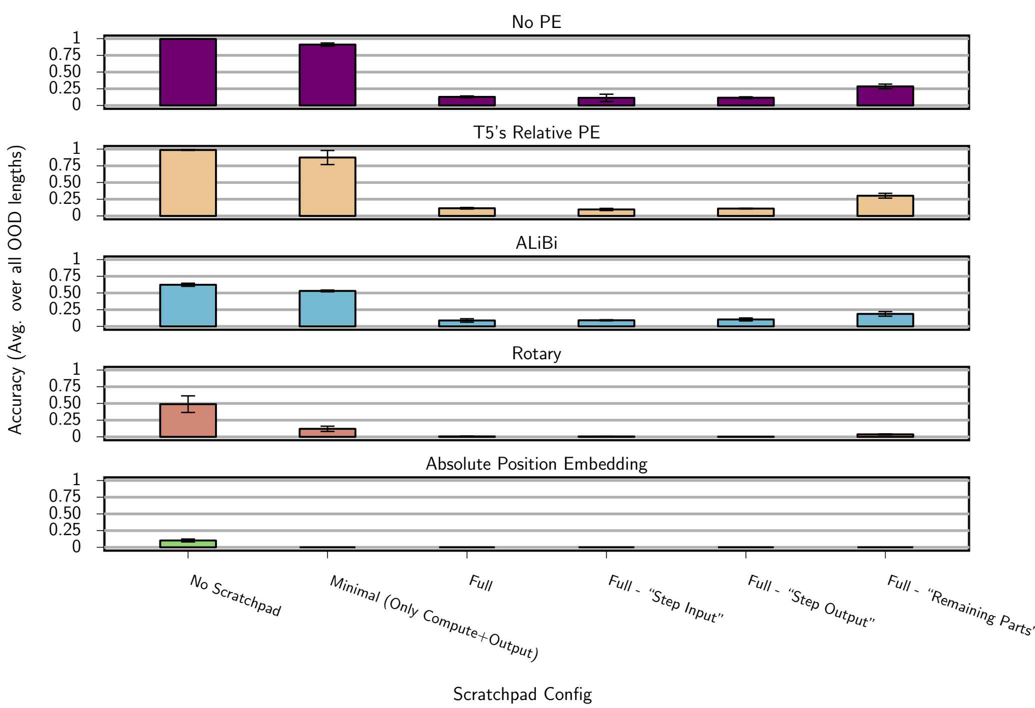

Figure E.10: Generalization of various scratchpad formats for each model on the **Sorting** task (Single Digit).

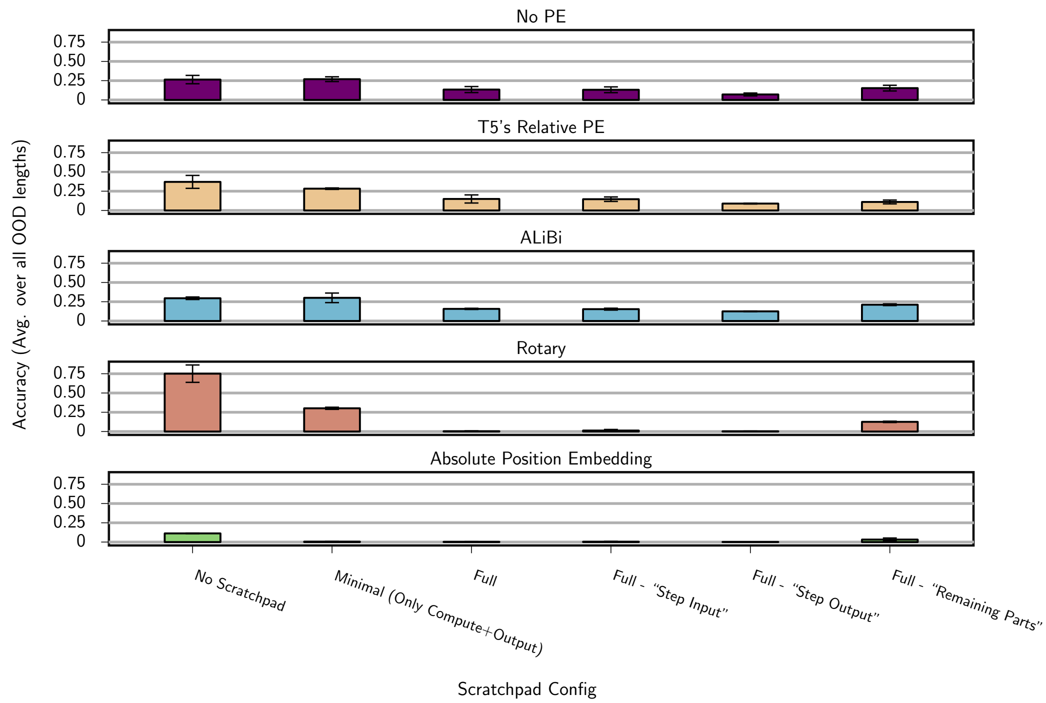

Figure E.11: Generalization of various scratchpad formats for each model on the **Sorting** task (Multi Digit).

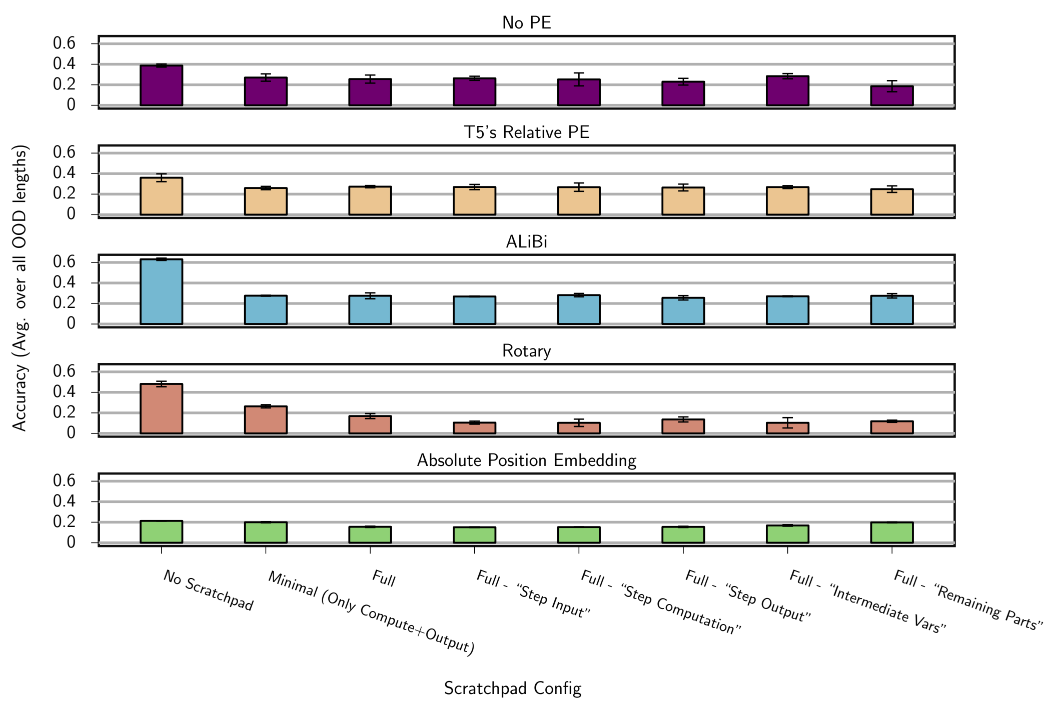

Figure E.12: Generalization of various scratchpad formats for each model on the **Polynomial Evaluation** task.

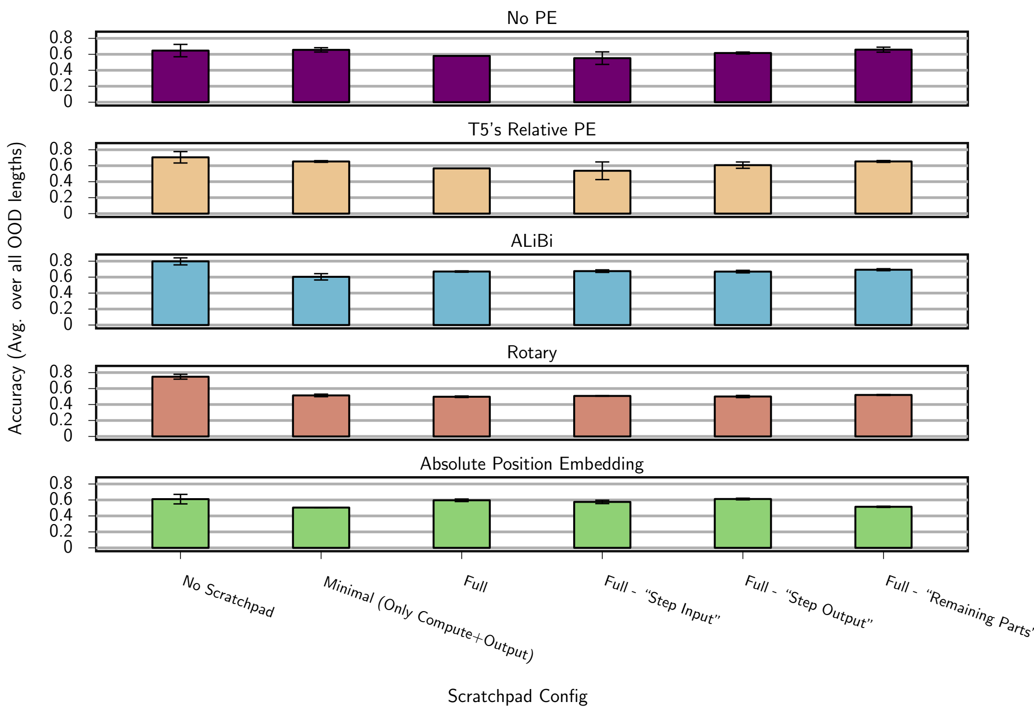

Figure E.13: Generalization of various scratchpad formats for each model on the **LEGO** task.