Dissecting Chain-of-Thought: A Study on Compositional In-Context Learning of MLPs

Anonymous Author(s)
Affiliation
Address
email

Abstract

Chain-of-thought (CoT) is a method that enables language models to handle complex reasoning tasks by decomposing them into simpler steps. Despite its success, the underlying mechanics of CoT are not yet fully understood. In an attempt to shed light on this, our study investigates the impact of CoT on the ability of transformers to in-context learn a simple to study, yet general family of compositional functions: multi-layer perceptrons (MLPs). In this setting, we reveal that the success of CoT can be attributed to breaking down in-context learning of a compositional function into two distinct phases: focusing on data related to each step of the composition and in-context learning the single-step composition function. Through both experimental and theoretical evidence, we demonstrate how CoT significantly reduces the sample complexity of in-context learning (ICL) and facilitates the learning of complex functions that non-CoT methods struggle with. Furthermore, we illustrate how transformers can transition from vanilla in-context learning to mastering a compositional function with CoT by simply incorporating an additional layer that performs the necessary filtering for CoT via the attention mechanism. In addition to these test-time benefits, we highlight how CoT accelerates pretraining by learning shortcuts to represent complex functions and how filtering plays an important role in pretraining. These findings collectively provide insights into the mechanics of CoT, inviting further investigation of its role in complex reasoning tasks.

1 Introduction

The advent of transformers [Vaswani et al., 2017] has revolutionized natural language processing, paving the way for remarkable performance in a wide array of tasks. LLMs, such as GPTs [Brown et al., 2020], have demonstrated an unparalleled ability to capture and leverage vast amounts of data, thereby facilitating near human-level performance across a variety of language generation tasks. Despite this success, a deep understanding of their underlying mechanisms remains elusive.

Chain-of-thought prompting [Wei et al., 2022c] is an emergent ability of transformers where the model solves a complex problem [Wei et al., 2022b], by decomposing it into intermediate steps. Intuitively, such this underlies the ability of general-purpose language models to accomplish previously-unseen complex tasks by leveraging more basic skills acquired during the pretraining phase.

Compositional learning and CoT has enjoyed significant recent success in practical language modeling tasks spanning question answering, code generation, and mathematical reasoning [Perez et al., 2021, Imani et al., 2023, Yuan et al., 2023]. In this work, we attempt to demystify some of the mechanics underlying this success and the benefits of CoT in terms of sample complexity and approximation ability. To do this we explore the role of CoT in learning in-context multi-layer perceptrons (MLP), that we believe can lead to a first set of insightful observations. Throughout, we ask:

Does CoT improve in-context learning of MLPs, and what are the underlying mechanics?

Submitted to 37th Conference on Neural Information Processing Systems (NeurIPS 2023). Do not distribute.

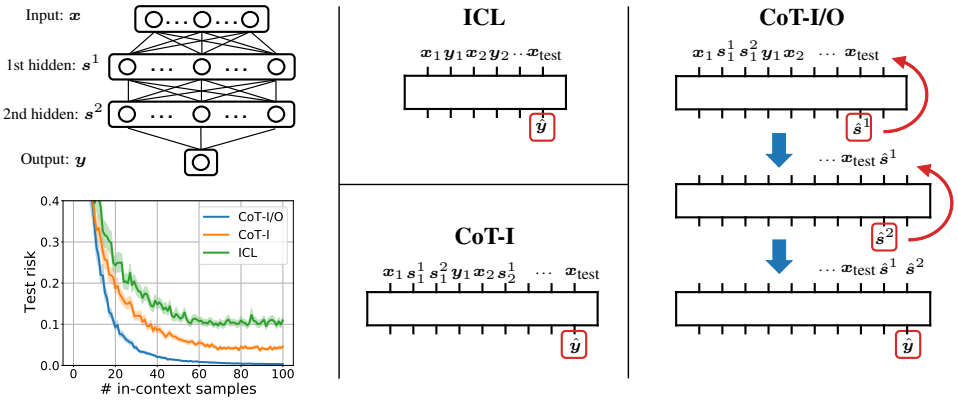


Figure 1: An illustration of ICL, CoT-I and CoT-I/O methods where we take 3-layer MLP as an example (see top left where $\boldsymbol{x}$, $\boldsymbol{y}$, $\boldsymbol{s}^1$, $\boldsymbol{s}^2$ represent input, output and hidden features). Then ICL takes in-context example of form $(\boldsymbol{x}, \boldsymbol{y})$ and makes prediction directly based on given $\boldsymbol{x}_{\text{test}}$, while CoT-I and CoT-I/O admit prompt with samples formed by $(\boldsymbol{x}, \boldsymbol{s}^1, \boldsymbol{s}^2, \boldsymbol{y})$. Differently, CoT-I/O makes recurrent predictions by feeding the intermediate output back to the input (as shown on the right). Bottom left shows the performances and more details are discussed in Sections 2 and 4.2.

Contributions: As our central contribution, we establish a rigorous and experimentally-supported abstraction that decouples CoT prompting into a *filtering phase* and an *in-context learning (ICL) phase*. In the *filtering phase*, the model attends to the relevant tokens within the prompt based on an instruction, and suppresses those irrelevant. In the *ICL phase*, the model runs inference on the filtered prompt to output a *step*. The model then moves to the next *step* in the chain. How a transformer architecture can actually realize this process is formalized in Theorem 1 for MLPs.

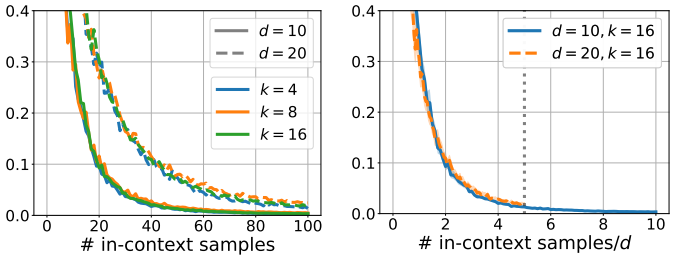
Building on this, we identify and thoroughly compare three schemes as illustrated in Figure 1. (a) ICL: In-context learning from input-output pairs provided in the prompt, (b) CoT-I: Examples in the prompt are augmented with intermediate steps, (c) CoT-I/O: The model also outputs intermediate steps during prediction. Our main contributions are:

- **Approximation and sample complexity:** Through experiments and theory, we establish that intermediate steps in CoT-I improves the sample complexity of learning whereas step-by-step output improves the approximation ability through looping. Specifically, CoT-I/O can learn an MLP with input dimension d and k neurons using $\mathcal{O}(\max(k, d))$ in-context samples by filtering individual layers and solving them via linear regression – in contrast to the $\Omega(kd)$ lower bound without step-augmented prompt. In line with theory, our experiments (e.g. Figs. 2&3) identify a striking universality phenomenon (as k varies) and also demonstrate clear approximation benefits of CoT compared to vanilla ICL.
- **Accelerated pretraining via learning shortcuts:** We construct deep linear MLPs where each layer is chosen from a discrete set of matrices. This is in contrast to above where MLP weights can be arbitrary. We show that CoT can dramatically accelerate pretraining by memorizing these discrete matrices and can infer all layers correctly from a *single* demonstration. Notably the pretraining loss goes to zero step-by-step where each step *"learns to filter a layer"*. Together, these showcase how CoT identifies composable shortcuts to avoid the need for solving linear regression. In contrast, we show that, ICL (without CoT) collapses to the linear regression performance as it fails to memorize exponentially many candidates (due to lack of composition).

The paper is organized as follows. In Section 2, we introduce the problem setup and preliminaries. Section 3 provides an empirical investigation of CoT with 2-layer MLPs and states our main theoretical results. Section 4 presents holistic experiments on the sample complexity and approximation benefits of CoT. Finally, we elucidate the benefits of CoT during pretraining via experiments on deep linear MLPs in Section 4.3. Related work and discussion are provided in Section 5 and 6.

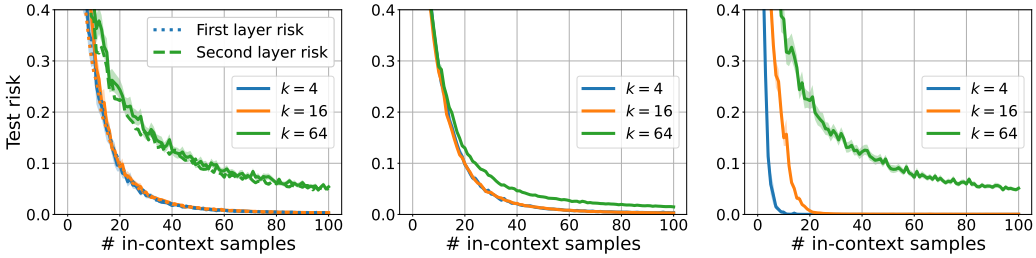
2 Preliminaries and Setup

Let $[n]$ denote set $\{1, 2 \cdots, n\}$. We will refer to vectors and matrices in bold text (e.g., $\boldsymbol{x}$, $\boldsymbol{A}$) while scalars will be represented in plain text (e.g., y). We will denote the input and output domains to be $\mathcal{X}$ and $\mathcal{Y}$ respectively (unless otherwise specified), and $\boldsymbol{x} \in \mathcal{X}$, $\boldsymbol{y} \in \mathcal{Y}$ denote the input and output.



(a) CoT-I/O: $d = 10$ *vs* 20 (b) Alignment in Fig. 2(a)

Figure 2: Solving 2-layer MLPs with different input dimension d and hidden neuron size k.



(a) CoT-I/O: composed risk ($d = 10$) (b) Risk of first layer (c) Risk of second layer

Figure 3: We decouple the composed risk of predicting 2-layer MLPs into risks of individual layers.

2.1 In-context Learning

Following Garg et al. [2022], the fundamental problem of vanilla in-context learning (ICL) , considers a prompt with input-output pairs of the following form:

$$\boldsymbol{p}_n(f) = (\boldsymbol{x}_i, \boldsymbol{y}_i)_{i=1}^n \quad \text{where} \quad \boldsymbol{y}_i = f(\boldsymbol{x}_i), \tag{P-ICL}$$

where the transition function $f \in \mathcal{F} : \mathcal{X} \to \mathcal{Y}$ remains the same within a single prompt but can vary across prompts; the subscript n represents the number of in-context samples contained in the prompt. Consider language translation as an example, then f is identified as the target language and the prompt can be defined as $\boldsymbol{p}(\texttt{Spanish})$ = ((*apple, manzana*), (*ball, pelota*), $\cdots$) or $\boldsymbol{p}(\texttt{French})$=((*cat, chat*), (*flower, fleur*), $\cdots$). Let TF denote any auto-regressive model (e.g., Decoder-only Transformer). Then, the problem of in-context learning aims to learn a model so that, given a prompt $\boldsymbol{p}$ and the test input $\boldsymbol{x}_{\text{test}}$, the TF can accurately predict the output such that

$$\texttt{TF}(\boldsymbol{p}_n(\tilde{f}), \boldsymbol{x}_{\text{test}}) \approx \tilde{f}(\boldsymbol{x}_{\text{test}}) \tag{2.1}$$

where $\tilde{f} \in \mathcal{F}$ is the test function which can be different to the functions used on training. Previous work [Zhou et al., 2022, Oymak et al.] has shown that typically, larger prompt size (n) improves the performance of the model.

2.2 Chain-of-thought Prompt and Prediction

As we already mentioned, the prompt in vanilla ICL (P-ICL) contains input-output pairs of the target function. This requires the model to learn the function $f \in \mathcal{F}$ in one shot, and as $\mathcal{F}$ becomes more complex, larger models and longer prompt length (n) are needed to make correct predictions (see green curves in Figures 4&5). Existing work for chain-of-thought methods [Wei et al., 2022c] observed that prompts containing step-by-step instructions help the model deconstruct the function and make better predictions. Specifically, consider a function composed from L subfunctions, that is $f := f_L \circ f_{L-1} \circ \dots f_1$. Then each intermediate output can be perceived as a step, and we can define a length-n CoT prompt corresponding to f with L steps (represented with $\boldsymbol{s}^\ell, \ell \in [L]$) as follows:

$$\boldsymbol{p}_n(f) = (\boldsymbol{x}_i, \boldsymbol{s}_i^1, \cdots \boldsymbol{s}_i^{L-1}, \boldsymbol{s}_i^L)_{i=1}^n \quad \text{where} \quad \boldsymbol{s}_i^\ell = f_\ell(\boldsymbol{s}_i^{\ell-1}),\ \ell \in [L]. \tag{P-CoT}$$

Here $\boldsymbol{x}_i = \boldsymbol{s}_i^0$, $\boldsymbol{y}_i = \boldsymbol{s}_i^L$ and $f_\ell \in \mathcal{F}_\ell$, $f \in \mathcal{F}_L \times \cdots \mathcal{F}_1 := \mathcal{F}$.

We now present two different ways that predictions can be made in the context of CoT:

CoT over input only (CoT-I). Compared to ICL, CoT-I takes in step-by-step instructions as inputs, which fragments the calculation of the target value into a set of simpler ones. However, the prediction

over the last token is performed all together in one shot. Our experimental results show that it helps in reducing sample complexity for TF to learn the function $\tilde{f}$ at hand (see orange curves in Figures 4&5). Then the CoT-I prediction is as in Eq. (2.1) but the prompt is given by (P-CoT).

$$\text{One-shot prediction: } \texttt{TF}(\boldsymbol{p}_n(\tilde{f}), \boldsymbol{x}_{\text{test}}) \approx \tilde{f}(\boldsymbol{x}_{\text{test}}). \tag{2.2}$$

CoT over both input and output (CoT-I/O). Even though CoT-I improves the sample complexity in calculating $\tilde{f}$, the TF still needs to be expressive enough as to approximate functions from the function class $\mathcal{F}$, since the prediction is made in one shot. To address this challenge, we consider the following scenario where in addition to applying CoT prompt, we make CoT predictions as well. Specifically, for a composed problem with inputs formed through (P-CoT), the model makes L-step predictions recurrently as follows:

$$\begin{aligned}
&\text{Step 1: } \texttt{TF}(\boldsymbol{p}_n(\tilde{f}), \boldsymbol{x}_{\text{test}}) := \hat{\boldsymbol{s}}^1 \\
&\text{Step 2: } \texttt{TF}(\boldsymbol{p}_n(\tilde{f}), \boldsymbol{x}_{\text{test}}, \hat{\boldsymbol{s}}^1) := \hat{\boldsymbol{s}}^2 \\
&\quad\vdots \\
&\text{Setp } L\text{: } \texttt{TF}(\boldsymbol{p}_n(\tilde{f}), \boldsymbol{x}_{\text{test}}, \hat{\boldsymbol{s}}^1 \cdots, \hat{\boldsymbol{s}}^{L-1}) \approx \tilde{f}(\boldsymbol{x}_{\text{test}}),
\end{aligned} \tag{2.3}$$

where at each step, model outputs an intermediate step ($\hat{\boldsymbol{s}}^\ell$) which will be feed back to the input sequence to make the next-step prediction ($\hat{\boldsymbol{s}}^{\ell+1}$). Following this strategy, the model only needs to learn the union of the sub-function sets, $\bigcup_{\ell=1}^{L} \mathcal{F}_\ell$, which scales linearly with the number of steps L. Blue curves in Figures 4 and 5 empirically reflect the benefits of CoT-I/O compared to ICL and CoT-I in improving sample efficiency and model expressivity.

3 Empirical and Theoretical Insights into CoT

In this section, we first investigate how CoT-I/O performs when learning 2-layer MLPs with input dimension d and hidden dimension k. Our experiments show that CoT-I/O requires only $O(\max(d, k))$ in-context samples. Then in Section 3.2, we provide our theoretical results showing that CoT-I/O can perform filtering over CoT prompt, and therefore, it turns to learn a 2-layer MLP as solving k d-dimensional ReLU and 1 k-dimensional linear regression problems.

3.1 Empirical Investigation of 2-layer MLPs

To investigate how MLP architecture will affect the CoT-I/O performance, we train 2-layer MLPs with different input dimension d and hidden layer size k, and results are shown in Figure 2 and 3 respectively. Let $\boldsymbol{x}, \boldsymbol{s}, y$ denote input, hidden state and output. We defer the implementation details to Section 4.2, and all the experiments are conductedwith small GPT-2 model.

CoT-I/O performance agnostic to k when $k \leq d$ (Figure 2). In Fig. 2(a), we train different MLPs with $d = 10, 20$ and $k = 4, 8, 16$. Solid and dashed curves respectively represent CoT-I/O test risk of $d = 10$ and 20 given different in-context samples. Results show that enlarging d will increase the samples needed in in-context learning, while performance keeps unchangable with varying $k \in \{4, 8, 16\}$. To further investigate how d impacts CoT-I/O accuracy, in Fig. 2(b), we rescale the horizontal axis by dividing with input dimension d, and put both $d = 10, k = 16$ (blue solid) and $d = 20, k = 16$ (orange dashed) results on it. The two curves are aligned, which demonstrates that the in-context sample complexity of CoT-I/O is linearly dependent on d.

Large k dominates CoT-I/O performance (Figure 3). Next, we investigate when k will turn to dominate the CoT-I/O performance. In Figure 3(a) we repeat the same experiments by fixing $d = 10$ but train with wider MLPs ($k = 64$), and blue, orange and green curves represent $k = 4, 16, 64$ results. Now since hidden dimension $k = 64$ is large, learning second layer requires more hidden features ($\boldsymbol{s}$), and therefore given $N = 100$ in-context samples (which will provide 100 $\boldsymbol{s}$s) is not enough to restore the second layer, and performance gaps appear between $k = 4, 16$ and $k = 64$. To quantify the existing gaps, we make single-step evaluations for both first layer and second layer, and results are shown in Fig. 3(b) and Fig. 3(c). Specifically, let $\boldsymbol{p}_n(\tilde{f})$ be a test prompt containing n in-context samples where $\tilde{f}$ represents arbitrary 2-layer MLP. Then given a test sample $(\boldsymbol{x}_{\text{test}}, \boldsymbol{s}_{\text{test}}, y_{\text{test}})$, the layer predictions are made as follows.

$$\begin{aligned}
&\text{1st layer prediction: } \texttt{TF}(\boldsymbol{p}_n(\tilde{f}), \boldsymbol{x}_{\text{test}}) := \hat{\boldsymbol{s}}, \\
&\text{2nd layer prediction: } \texttt{TF}(\boldsymbol{p}_n(\tilde{f}), \boldsymbol{x}_{\text{test}}, \boldsymbol{s}_{\text{test}}) := \hat{y},
\end{aligned}$$

the test risks are calculated by $\|\hat{\boldsymbol{s}} - \boldsymbol{s}_{\text{test}}\|^2$ and $(\hat{y} - y_{\text{test}})^2$ (The risks shown in the figures are normalized so that they are comparable. Also check Section 4.2 and appendix for more details). Evidences in Fig. 3(b) and 3(c) show that enlarging k will not influence the first layer prediction, but increase the number of samples required in learning the second layer. What's more, we also plot the first layer risks of $k = 4, 16$ (blue/orange dotted) and second layer risk of $k = 64$ (green dashed) in Fig. 3(a), and they are aligned with CoT-I/O composed risks, which verifies that CoT-I/O learns 2-layer MLP as compositional learning separate layers.

3.2 Provable Approximation of MLPs via Chain-of-Thought

The observations we made in Section 3.1 are indicative of the model processing each one of the two layers sequentially. Now in this section, we state our main contribution of establishing a result that decouples CoT-based in-context learning (CoT-I/O) into two phases: (1) *Filtering Phase:* Given a prompt that contains features of multiple MLP layers, retrieve only the features related to a target layer to create a *homogeneous* prompt. (2) *ICL Phase:* Given filtered prompt, learn the target layer weights through gradient descent. Combining these two phases, and looping over all layers, we will show that there exists a transformer architecture such that CoT-I/O can provably approximate a multilayer MLP up to a given resolution. To state our result, we assume access to an oracle that performs linear regression and consider the consider the condition number of the data matrix.

Definition 1 (MLP and condition number) *Consider a multilayer MLP defined by the recursion* $\boldsymbol{s}_i^\ell = \phi(\boldsymbol{W}_\ell \boldsymbol{s}_i^{\ell-1})$ *for* $\ell \in [L]$, $i \in [n]$ *and* $\boldsymbol{s}_i^0 = \boldsymbol{x}_i$. *Here* $\phi(x) = \max(\alpha x, x)$ *is a Leaky-ReLU activation with* $1 \geq \alpha > 0$. *Define the feature matrix* $\boldsymbol{T}_\ell = [\boldsymbol{s}_1^\ell \ \dots \ \boldsymbol{s}_n^\ell]^\top$ *and define its condition number* $\kappa_\ell = \sigma_{\max}(\boldsymbol{T}_\ell)/\sigma_{\min}(\boldsymbol{T}_\ell)$ *(with* $\sigma_{\min} := 0$ *for fat matrices) and* $\kappa_{\max} = \max_{0 \leq \ell < L} \kappa_\ell$.

Assumption 1 (Oracle Model) *We assume access to a transformer* $\texttt{TF}_{LR}$ *which can run* T *steps of gradient descent on the quadratic loss* $\mathcal{L}(\boldsymbol{w}) = \sum_{i=1}^n (y_i - \boldsymbol{w}^\top \boldsymbol{x}_i)^2$ *given a prompt of the form* $(\boldsymbol{x}_1, y_1, \dots, \boldsymbol{x}_n, y_n)$.

We remark that this assumption is realistic and has been formally established by earlier work [Giannou et al., 2023, Akyürek et al., 2022]. Our CoT abstraction builds on these to demonstrate that CoT-I/O can call a blackbox TF model to implement a compositional function when combined with filtering.

The following result summarizes our main theoretical contribution. The precise statement is deferred to the supplementary material.

Theorem 1 (Decoupling CoT) *Consider a prompt* $\boldsymbol{p}_n(f)$ *generated from an* L*-layer MLP* $f(\cdot)$ *as described in Definition 1, and assume given test example* $(\boldsymbol{x}_{test}, \boldsymbol{s}_{test}^1, \dots \boldsymbol{s}_{test}^L)$. *For any resolution* $\epsilon > 0$, *there exists* $\delta = \delta(\epsilon)$, *iteration choice* $T = \mathcal{O}(\kappa_{\max}^2 \log(1/\epsilon))$, *and a backend transformer construction* $\texttt{TF}_{BE}$ *such that the concatenated transformer* $\texttt{TF} = \texttt{TF}_{LR} \circ \texttt{TF}_{BE}$ *implements the following: Let* $(\hat{\boldsymbol{s}}^i)_{i=1}^{\ell-1}$ *denote the first* $\ell - 1$ *CoT-I/O outputs of* TF *and set* $\boldsymbol{p}[\ell] = (\boldsymbol{p}_n(f), \boldsymbol{x}_{test}, \hat{\boldsymbol{s}}^1 \dots \hat{\boldsymbol{s}}^{\ell-1})$. *At step* ℓ, TF *implements*

1. ***Filtering.*** *Define the filtered prompt with input/output features of layer* ℓ,

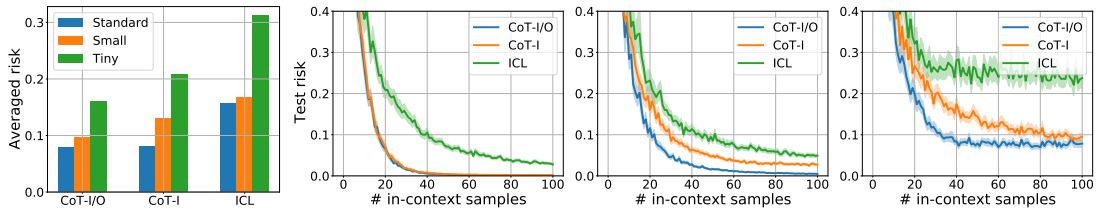
$$\boldsymbol{p}_n^{filter} = \begin{pmatrix} \dots \mathbf{0}, \boldsymbol{s}_1^{\ell-1}, & \mathbf{0} \ \dots \mathbf{0}, \boldsymbol{s}_n^{\ell-1}, & \mathbf{0} \ \dots \mathbf{0}, \hat{\boldsymbol{s}}^{\ell-1} \\ \dots \mathbf{0}, & \mathbf{0}, & \boldsymbol{s}_1^\ell \dots \mathbf{0}, & \mathbf{0}, & \boldsymbol{s}_n^\ell \dots \mathbf{0}, & \mathbf{0} \end{pmatrix}.$$

   There exists a fixed projection matrix $\boldsymbol{\Pi}$ *that applies individually on tokens such that the backend output obeys* $\|\boldsymbol{\Pi}(\texttt{TF}_{BE}(\boldsymbol{p}[\ell])) - \boldsymbol{p}_n^{filter}\| \leq \delta$.

2. ***Gradient descent.*** *The combined model obeys* $\|\texttt{TF}(\boldsymbol{p}[\ell]) - \boldsymbol{s}_{test}^\ell\| \leq \ell \cdot \epsilon / L$.

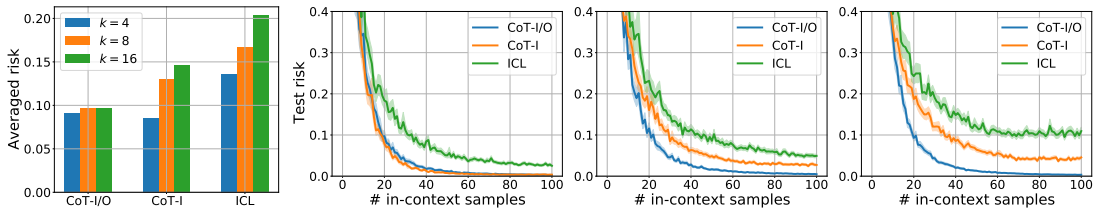
$\texttt{TF}_{BE}$ *has constant number of layers independent of* T *and* n. *Consequently, after* L *rounds of CoT-I/O,* TF *outputs* $f(\boldsymbol{x}_{test})$ *up to* ϵ *accuracy.*

Remark 1 *Note that, this result effectively shows that, with a sufficiently good blackbox transformer* $\texttt{TF}_{LR}$ *(per Assumption 1), CoT-I/O can learn an* L*-layer MLP using in-context sample size* $n > \max_{\ell \in [L]} d_\ell$ *where* d_ℓ *is the input dimension of* ℓ*th layer. This is assuming condition number* $\kappa_{\max}$ *of the problem is finite as soon as all layers become over-determined. Consequently, CoT-I/O needs* $\max(k, d)$ *sample complexity to learn a two layer MLP. This provides a formal justification for the observation that empirical CoT-I/O performance is agnostic to* k *as long as* $k \leq d$.



(a) Averaged risk (b) Standard GPT-2 (c) Small GPT-2 (d) Tiny GPT-2

Figure 4: Compare the three methods in solving 2-layer MLPs using different GPT-2 models.



(a) Averaged risk (b) $k = 4$ (c) $k = 8$ (d) $k = 16$

Figure 5: Compare the three methods in solving 2-layer MLPs of different hidden sizes.

We provide the filtering statements in the Appendix, and the key components of our construction are the following: (i) Inputs are projected through the embedding layer in which a set of encodings, an enumeration of the tokens $(1, 2, \ldots, N)$, an enumeration of the layers $(1, 2, \ldots, L)$ and an identifier for each layer already predicted are all attached. Notice that this "modification" to the input only depends on the sequence length and is agnostic to the token to-be-predicted. This allows for an automated looping over L predictions. (ii) We use this information to extract the sequence length N and the current layer ℓ to-be-predicted. (iii) With these at hand we construct an 'if - then' type of function using the ReLU layers to filter out the samples that are not needed for the prediction.

4 Experimental Results

4.1 Model Training

In Figure 1 and Section 2, we have discussed the vanilla ICL, CoT-I and CoT-I/O methods. Intuitively, ICL is a special case of CoT-I (or CoT-I/O) if assuming only one step is performed. Therefore, we will implement CoT-I and CoT-I/O for model training in the following.

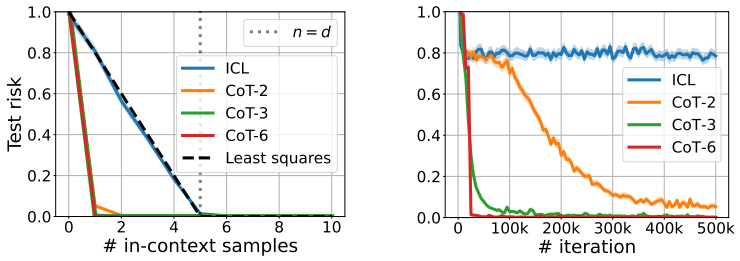
Consider the CoT prompt (P-CoT), and assume that $\boldsymbol{x} \sim \mathcal{D}_{\mathcal{X}}$, and $f_\ell \sim \mathcal{D}_\ell, \ell \in [L]$, where L is the number of compositions/steps so that the final prediction should approximate $f(\boldsymbol{x}) = f_L(f_{L-1} \ldots f_1(\boldsymbol{x})) := \boldsymbol{y} \in \mathcal{Y}$. Define $\ell(\hat{\boldsymbol{y}}, \boldsymbol{y}) : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$ be a loss function, and for simplicity, assume $f_\ell(\ldots f_1(\boldsymbol{x})) \in \mathcal{Y}, \ell \in [L]$. Let N be in-context window of TF, that is, TF can only admit prompt containing up to N in-context samples. Generally, our goal is to ensure high prediction performance given any length-n prompt, where $n \in [N]$. To this end, we train the model using prompts with length from 1 to N equally and aim to minimize the averaged risk over different prompt size. Assume model TF is parameterized by $\boldsymbol{\theta}$ and consider meta learning problem. Then the objective functions for CoT-I and CoT-I/O are defined as follows.

$$\hat{\boldsymbol{\theta}}^{\text{CoT-I}} = \arg\min_{\boldsymbol{\theta}} \mathbb{E}_{(\boldsymbol{x}_n)_{n=1}^N, (f_\ell)_{\ell=1}^L} \left[\frac{1}{N} \sum_{n=1}^{N} \ell(\hat{\boldsymbol{y}}_n, f(\boldsymbol{x}_n)) \right] \quad \text{where} \quad \hat{\boldsymbol{y}}_n = \texttt{TF}(\boldsymbol{p}_n(f), \boldsymbol{x}_n)$$

and

$$\hat{\boldsymbol{\theta}}^{\text{CoT-I/O}} = \arg\min_{\boldsymbol{\theta}} \mathbb{E}_{(\boldsymbol{x}_n)_{n=1}^N, (f_\ell)_{\ell=1}^L} \left[\frac{1}{NL} \sum_{n=1}^{N} \sum_{\ell=1}^{L} \ell(\hat{\boldsymbol{s}}_n^\ell, \boldsymbol{s}_n^\ell) \right] \quad \text{where} \quad \hat{\boldsymbol{s}}_n^\ell = \texttt{TF}(\boldsymbol{p}_n(f), \boldsymbol{x}_n \cdots \boldsymbol{s}_n^{\ell-1}).$$

Here $\boldsymbol{p}_n(f)$ is given by (P-CoT), and same as mentioned above, $\boldsymbol{s}^0 = \boldsymbol{x}$ and $\boldsymbol{s}^L = \boldsymbol{y}$. All $\boldsymbol{x}$ and f_ℓ are independent and we take the expectation of the risk over their specific distributions.



(a) Point-to-point meta prediction (b) One-shot prediction over time

Figure 6: Evaluations over deep linear MLPs using CoT-I/O and ICL where CoT-X represents the X-step CoT-I/O. Fig. 6(a) shows point-to-point meta results where model is trained with sufficiently many samples, and in Fig. 6(b), we make prediction during training and evaluate the performance given only one in-context sample. See Section 4.3 for implementation details.

4.2 2-layer Random MLPs

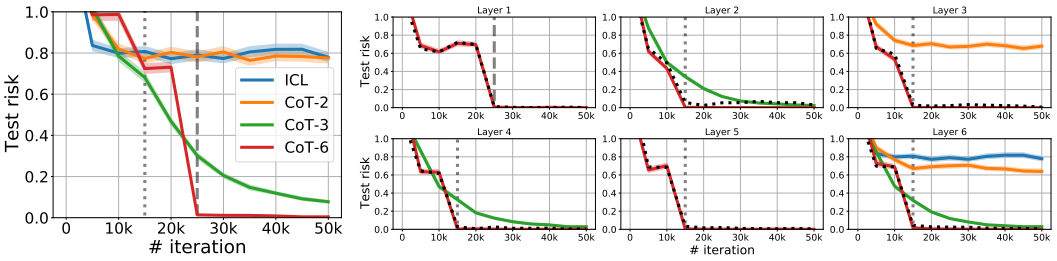
For a clear exposition, we first focus on the case of two layer MLPs which is 2-step tasks and compare three different methods: ICL, CoT-I and CoT-I/O. Comparison results are shown in Figures 4 and 5.

Dataset. Consider 2-layer MLPs with input $\boldsymbol{x} \in \mathbb{R}^d$, hidden feature (step-1 output) $\boldsymbol{s} \in \mathbb{R}^k$, and output $y \in \mathbb{R}$, where $\boldsymbol{s} = f_1(\boldsymbol{x}) := (\boldsymbol{W}\boldsymbol{x})_+$ and $y = f_2(\boldsymbol{s}) := \boldsymbol{v}^\top \boldsymbol{s}$. Here $\boldsymbol{W} \in \mathbb{R}^{k\times d}$, $\boldsymbol{v} \in \mathbb{R}^k$ are the first and second layer/sub-function parameters, and $(x)_+ = \max(x, 0)$ is ReLU activation. Hence, the function is composed as $y = \boldsymbol{v}^\top(\boldsymbol{W}\boldsymbol{x})_+$. We define the function distributions as follows: each entry of $\boldsymbol{W}$ is sampled via $\boldsymbol{W}_{ij} \sim \mathcal{N}(0, \frac{2}{k})$, and $\boldsymbol{v} \sim \mathcal{N}(0, \boldsymbol{I}_k)$, and inputs are sampled randomly through $\boldsymbol{x} \sim \mathcal{N}(0, \boldsymbol{I}_d)$[1]. We apply quadratic loss in our experiments. To avoid the implicit bias due to distribution shift, both training and test datasets are generated following the same strategy.

Varying model sizes (Figure 4). We first evaluate the benefits of CoT-I/O from ICL and CoT-I over different TF models. Fix $d = 10$ and $k = 8$, and train over three different GPT-2 models: standard, small and tiny GPT-2. The small GPT-2 has 6 layers, 4 attention heads per layer and 128 dimensional embeddings. While, standard GPT-2 contains twice of the layer number, attention heads and embedding dimension of small GPT-2 and tiny GPT-2 contains only half of each. We set $N = 100$, and test the performance given prompts with n in-context samples (n from 1 to 100), and test risks of each given n are shown in Figs. 4(b), 4(c) and 4(d). The blue, orange and green curves stand for CoT-I/O, CoT-I and ICL respectively. In Fig. 4(a), we display the averaged risks. Results show that using CoT-I/O, small GPT-2 can solve 2-layer MLPs at hand using 60 samples, and CoT-I needs standard GPT-2. However, ICL can not solve the problem since it does not achieve zero test risk using standard GPT-2 model and given up to 100 samples. One interpretation is that: to learn the 2-layer MLPs in one shot, ICL needs at least $O(dk + d)$ samples to restore all function parameters. Different to ICL, CoT-I and CoT-I/O have implicit samples contained in the CoT prompt. Let $f_1 \in \mathcal{F}_1$ (first layer) and $f_2 \in \mathcal{F}_2$ (second layer). Comparing the performance of CoT-I and CoT-I/O shows that, standard GPT-2 is able to learn the composed function $f = f_2 \circ f_1 \in \mathcal{F}$, which small GPT-2 cannot express.

Varying MLP widths (Figure 5). Next, we investigate how different MLP widths will impact the performance (by varying hidden neuron size $k \in \{4, 8, 16\}$) and results are shown in Figure 5. Blue, orange and green curves in Figs. 5(b), 5(c) and 5(d) correspond to the hidden layer size $k = 4$, 8, and 16 respectively, and Fig. 5(a) presents the averaged risks. Fix $d = 10$, $N = 100$ and train with small GPT-2. As we have discussed in Section 3, CoT-I/O can learn a 2-layer MLP using around 60 samples for all $k = 4, 8, 16$ thanks to its ability of deconstructing the composed functions. While, CoT-I can only learn the narrow MLPs with $k = 4$ and ICL is not able to learn any of it. What's more, we also observe that performance of CoT-I and CoT-I/O differs a lot with varying k (e.g., see averaged risks in Fig. 5(a)), and it can be explained as: enlarging k results in more complex $\mathcal{F}_1$ and $\mathcal{F}_2$, and therefore learning $\mathcal{F} = \mathcal{F}_2 \times \mathcal{F}_1$ becomes more challenging for ICL and CoT-I.

[1]Following this data generation strategy, $\boldsymbol{x}$, $\boldsymbol{s}$ and y have the same expected norm, and risk curves shown in the plots are normalized.



(a) Composed risk (b) Risk of each layer output

Figure 7: Fig. 7(a) is generated by magnifying the initial 50k iterations of Fig. 6(b), and we decouple the composed risks from predicting 6-layer linear MLPs into each layer prediction and results are depicted in Fig. 7(b). Check Section 4.3 for implementation details.

4.3 Deep Linear MLPs

In Sections 3.1 and 4.2, we have discussed the approximation benefit of CoT-I/O and how it in-context learns 2-layer random MLPs by parallelly learning k d-dimensional ReLU and 1 k-dimensional linear regression, and composing them. In this section, we investigate the behaviors of CoT-I/O in learning longer compositions. For cleaner notation, in the following we use CoT to refer to as CoT-I/O.

Dataset. Consider a L-layer linear MLPs with input $\boldsymbol{x} \in \mathbb{R}^d \sim \mathcal{N}(0, \boldsymbol{I}_d)$, and output generated by $\boldsymbol{y} = \boldsymbol{W}_L \boldsymbol{W}_{L-1} \cdots \boldsymbol{W}_1 \boldsymbol{x}$, where $\boldsymbol{W}_\ell \in \mathbb{R}^{d \times d}$ is the ℓth layer parameters, $\ell \in [L]$. In this work, to better understand the emergent ability of CoT, we assume each layer is functionalized from the same discrete sub-function set $\bar{\mathcal{F}} = \{\bar{\boldsymbol{W}}_k : \bar{\boldsymbol{W}}_k^\top \bar{\boldsymbol{W}}_k = \boldsymbol{I}, k \in [K]\}$[2]. Therefore to learn the L-layer neural net, CoT only needs to learn $\bar{\mathcal{F}}$ with $|\bar{\mathcal{F}}| = K$, whereas ICL needs to learn function set $\bar{\mathcal{F}}^L$, which contains K^L random matrices.

Composition ability of CoT (Figures 6). Set $d = 5$, $L = 6$ and $K = 4$. At each round, randomly pick L matrices $\boldsymbol{W}_\ell$, $\ell \in [L]$ from $\bar{\mathcal{F}}$ so that for any input $\boldsymbol{x}$, we can form a chain

$$\boldsymbol{x} \to \boldsymbol{s}^1 \to \boldsymbol{s}^2 \cdots \to \boldsymbol{s}^6 := \boldsymbol{y},$$

where $\boldsymbol{s}^\ell = \boldsymbol{W}_\ell \boldsymbol{s}^{\ell-1}$, $\ell \in [L]$ and $\boldsymbol{s}^0 := \boldsymbol{x}$. Let CoT-$X$ denote X-step CoT-I/O method. For example, the in-context sample of CoT-6 has form $(\boldsymbol{x}, \boldsymbol{s}^1, \boldsymbol{s}^2, \ldots \boldsymbol{s}^5, \boldsymbol{y})$, which contains all the intermediate information outputted by each layer; while CoT-3, CoT-2 have prompt samples formed as $(\boldsymbol{x}, \boldsymbol{s}^2, \boldsymbol{s}^4, \boldsymbol{y})$ and $(\boldsymbol{x}, \boldsymbol{s}^3, \boldsymbol{y})$ respectively. In this setting, ICL can also be named as CoT-1, whose prompt contains $(\boldsymbol{x}, \boldsymbol{y})$ pairs. Intuitively, to solve the length-6 chain, CoT-X needs to learn a model that remembers $4^{6/X}$ matrices. Therefore, ICL is hard to solve the problem since it needs to remember $4^6 = 4,096$ matrices (all the combinations of the 4 matrices we use for training and testing) compared to the 4 for CoT-6.

We train small GPT-2 models using all the CoT-2/-3/-6 and ICL methods and results are presented in Fig. 6(a). From this figure, we can see that CoT-2 (orange), CoT-3 (green) and CoT-6 (red) performance curves are overlapping, and all can make precise predictions in one shot (given in-context example $n = 1$). It seems that TF has exactly learned to remember up to 64 matrices (for CoT-2) and compose up to 6 layers (for CoT-6). However, ICL (blue) is not able to learn the 6-layer MLPs in one shot. In black dashed curve, we solve linear regression $y = \boldsymbol{\beta}^\top \boldsymbol{x}$ directly via least squares given n random training samples, where $\boldsymbol{x}$ is the input and y is from the output of the 6-layer MLPs (e.g., $\boldsymbol{y}[0]$), and plot the test risks given $n = 1, \ldots 10$. From Fig. 6(a) we can see that ICL curve is aligned with least squares performance, which implies that, instead of remembering all $4,096$ matrices, ICL turns to solve the problem from the linear regression phase.

In addition to the meta-learning results which show the approximation benefits of multi-step CoT, we also investigate the convergence rate of CoT-2/-3/-6 and ICL, and results are displayed in Fig. 6(b). We test the one-shot performance during training and results show that CoT-6 converges fastest, since it has the smallest sub-function set, and given the same tasks (e.g., deep neural nets), shortening the chain results in slower convergence. It evidences us that taking more steps helps in learning complex problems faster and better.

[2]Here, we create unitary matrices to make sure the norm of feature is kept constant ($\|\boldsymbol{x}\| = \|\boldsymbol{y}\| = \|\boldsymbol{s}^\ell\|$), so that evaluations over different layers are fair.

Evidence of Filtering (Figure 7). In Theorem 1 and appendix, we state that transformers can perform filtering over CoT prompts, and 2-layer MLP results are aligned with our theoretical findings. But can we actually capture filtering behaviors? In Fig. 7(a), we display the results of the first 50k iterations from Fig. 6(b), and observe that there exist risk drops in CoT-6 (red) at 15k and 25k'th iteration (shown in gray dotted and dashed lines). Then in the Fig. 7(b), we plot the test risk of each layer prediction (by feeding the model with correct intermediate features not the predicted ones), and CoT-6 (red) predicts the outputs from all 6 layers ($\boldsymbol{s}^1, \cdots, \boldsymbol{s}^L$). From this figure, we can find that there exist risk drops as well when predicting different layers, which appear at iteration either 15k (for layer 2, 3, 4, 5, 6) or 25k (for layer 1). This implies that the model learns to predict each step/layer function separately. Furthermore, we test the filtering evidence of ℓth layer by filling irrelevant positions with random features. Specifically, in-context example is formed by

$$(\boldsymbol{z}^0, \cdots, \boldsymbol{s}^{\ell-1}, \boldsymbol{s}^{\ell}, \boldsymbol{z}^{\ell+1}, \dots \boldsymbol{z}^L), \quad \text{where } \boldsymbol{s}^{\ell} = \boldsymbol{W}_{\ell}(\boldsymbol{s}^{\ell-1}) \text{ and } \boldsymbol{z} \sim \mathcal{N}(0, \boldsymbol{I}_d).$$

Test risks are displayed in black dotted curves in Fig. 7(b) and they are all aligned with the CoT-6 curves (red). Each layer's prediction only focuses on the corresponding intermediate steps in the prefix and ignores the irrelevant features, which evidences that filtering is indeed performed.

5 Related Work

With the success of LLMs and prompt structure [Lester et al., 2021], there is growing interest in in-context learning (ICL) from both theoretical and experimental lens [Garg et al., 2022, Brown et al., 2020, von Oswald et al., 2022, Dai et al., 2022, Min et al., 2022, Lyu et al., 2022, Li et al., 2023, Xie et al., 2021, Min et al., 2021, Wei et al., 2023]. As an extension, chain-of-thought (CoT) prompts have made impressive improvements in performing complex reasoning by decomposing it into step-by-step intermediate solutions [Wei et al., 2022c, Narang et al., 2020, Lampinen et al., 2022, Wei et al., 2022b, Zhou et al., 2022, Nye et al., 2021, Veličković and Blundell, 2021, Lanchantin et al., 2023], which in general, shows the ability of transformer in solving compositional functions, and the idea of learning how to compose skills has been well studied in other literatures [Sahni et al., 2017, Liška et al., 2018]. More specifically, for the problem of learning shallow networks, there are several well known hardness results Goel et al. [2017, 2020], Zhang et al. [2019]. In particular, Hahn and Goyal [2023] shows a formal learnability bound which implies that compositional structure can benefit ICL. However, most of the work focuses on investigating empirical benefits and algorithmic designing of CoT, and there exists little effort studying the underlying mechanisms of CoT.

Considering the expressivity of the transformer architecture itself, Yun et al. [2019] showed that TFs are universal sequence to sequence approximators. More recently, Giannou et al. [2023] use an explicit construction to show that shallow TFs can be used to run general purpose programs as long as we loop them. Other works have also shown the turing-completeness of the TF architecture but these typically require infinite/high precision and recursion around attention layers [Wei et al., 2022a, Pérez et al., 2019, 2021, Liu et al., 2022]. Closer to our work, Akyürek et al. [2022] prove that a transformer with constant number of layers can implement gradient descent in solving linear regression, and Giannou et al. [2023] introduce similar results by looping outputs back into inputs. In this work, we prove CoT can be treated as: first apply filtering on the CoT prompts using special construction, and then in-context learn the filtered prompt.

6 Conclusion and Discussion

In this work, we investigate chain-of-thought prompting and shed light on how it enables compositional learning of multilayer perceptrons step-by-step. Specially, we have explored and contrasted three methods: ICL, CoT-I and CoT-I/O, and found that CoT-I/O facilitates better approximation and faster convergence through looping and sample efficiency. Additionally, we empirically and theoretically demonstrated that to learn a 2-layer MLP with d-dimensional input and k neurons, CoT-I/O requires $\mathcal{O}(\max(d, k))$ in-context samples whereas ICL runs into approximation error bottlenecks.

There are several interesting avenues for future research building on our findings. To what extent our decoupling of CoT (filtering followed by ICL) align with the empirical evidence in practical problems such as code generation and mathematical reasoning? We have shown that CoT-I/O can rely on linear regression oracle to learn an MLP. To what extent transformers can approximate MLPs without CoT-I/O (e.g. with CoT-I), what are lower/upper bounds?

References

Ekin Akyürek, Dale Schuurmans, Jacob Andreas, Tengyu Ma, and Denny Zhou. What learning algorithm is in-context learning? investigations with linear models. *arXiv preprint arXiv:2211.15661*, 2022.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

Damai Dai, Yutao Sun, Li Dong, Yaru Hao, Zhifang Sui, and Furu Wei. Why can gpt learn in-context? language models secretly perform gradient descent as meta optimizers. *arXiv preprint arXiv:2212.10559*, 2022.

Shivam Garg, Dimitris Tsipras, Percy S Liang, and Gregory Valiant. What can transformers learn in-context? a case study of simple function classes. *Advances in Neural Information Processing Systems*, 35:30583–30598, 2022.

Angeliki Giannou, Shashank Rajput, Jy-yong Sohn, Kangwook Lee, Jason D Lee, and Dimitris Papailiopoulos. Looped transformers as programmable computers. *arXiv preprint arXiv:2301.13196*, 2023.

Surbhi Goel, Varun Kanade, Adam Klivans, and Justin Thaler. Reliably learning the relu in polynomial time. In *Conference on Learning Theory*, pages 1004–1042. PMLR, 2017.

Surbhi Goel, Adam Klivans, Pasin Manurangsi, and Daniel Reichman. Tight hardness results for training depth-2 relu networks. *arXiv preprint arXiv:2011.13550*, 2020.

Michael Hahn and Navin Goyal. A theory of emergent in-context learning as implicit structure induction. *arXiv preprint arXiv:2303.07971*, 2023.

Shima Imani, Liang Du, and Harsh Shrivastava. Mathprompter: Mathematical reasoning using large language models. *arXiv preprint arXiv:2303.05398*, 2023.

Andrew K Lampinen, Ishita Dasgupta, Stephanie CY Chan, Kory Matthewson, Michael Henry Tessler, Antonia Creswell, James L McClelland, Jane X Wang, and Felix Hill. Can language models learn from explanations in context? *arXiv preprint arXiv:2204.02329*, 2022.

Jack Lanchantin, Shubham Toshniwal, Jason Weston, Arthur Szlam, and Sainbayar Sukhbaatar. Learning to reason and memorize with self-notes. *arXiv preprint arXiv:2305.00833*, 2023.

Brian Lester, Rami Al-Rfou, and Noah Constant. The power of scale for parameter-efficient prompt tuning. *arXiv preprint arXiv:2104.08691*, 2021.

Yingcong Li, M Emrullah Ildiz, Dimitris Papailiopoulos, and Samet Oymak. Transformers as algorithms: Generalization and implicit model selection in in-context learning. *arXiv preprint arXiv:2301.07067*, 2023.

Adam Liška, Germán Kruszewski, and Marco Baroni. Memorize or generalize? searching for a compositional rnn in a haystack. *arXiv preprint arXiv:1802.06467*, 2018.

Bingbin Liu, Jordan T Ash, Surbhi Goel, Akshay Krishnamurthy, and Cyril Zhang. Transformers learn shortcuts to automata. *arXiv preprint arXiv:2210.10749*, 2022.

Xinxi Lyu, Sewon Min, Iz Beltagy, Luke Zettlemoyer, and Hannaneh Hajishirzi. Z-icl: Zero-shot in-context learning with pseudo-demonstrations. *arXiv preprint arXiv:2212.09865*, 2022.

Sewon Min, Mike Lewis, Luke Zettlemoyer, and Hannaneh Hajishirzi. Metaicl: Learning to learn in context. *arXiv preprint arXiv:2110.15943*, 2021.

Sewon Min, Xinxi Lyu, Ari Holtzman, Mikel Artetxe, Mike Lewis, Hannaneh Hajishirzi, and Luke Zettlemoyer. Rethinking the role of demonstrations: What makes in-context learning work? *arXiv preprint arXiv:2202.12837*, 2022.

Sharan Narang, Colin Raffel, Katherine Lee, Adam Roberts, Noah Fiedel, and Karishma Malkan. Wt5?! training text-to-text models to explain their predictions. *arXiv preprint arXiv:2004.14546*, 2020.

Maxwell Nye, Anders Johan Andreassen, Guy Gur-Ari, Henryk Michalewski, Jacob Austin, David Bieber, David Dohan, Aitor Lewkowycz, Maarten Bosma, David Luan, et al. Show your work: Scratchpads for intermediate computation with language models. *arXiv preprint arXiv:2112.00114*, 2021.

Samet Oymak, Ankit Singh Rawat, Mahdi Soltanolkotabi, and Christos Thrampoulidis. On the role of attention in prompt-tuning. In *ICLR 2023 Workshop on Mathematical and Empirical Understanding of Foundation Models*.

Jorge Pérez, Javier Marinković, and Pablo Barceló. On the turing completeness of modern neural network architectures. *arXiv preprint arXiv:1901.03429*, 2019.

Jorge Pérez, Pablo Barceló, and Javier Marinkovic. Attention is turing complete. *The Journal of Machine Learning Research*, 22(1):3463–3497, 2021.

Luis Perez, Lizi Ottens, and Sudharshan Viswanathan. Automatic code generation using pre-trained language models. *arXiv preprint arXiv:2102.10535*, 2021.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

Himanshu Sahni, Saurabh Kumar, Farhan Tejani, and Charles Isbell. Learning to compose skills. *arXiv preprint arXiv:1711.11289*, 2017.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Petar Veličković and Charles Blundell. Neural algorithmic reasoning. *Patterns*, 2(7):100273, 2021.

Johannes von Oswald, Eyvind Niklasson, Ettore Randazzo, João Sacramento, Alexander Mordvintsev, Andrey Zhmoginov, and Max Vladymyrov. Transformers learn in-context by gradient descent. *arXiv preprint arXiv:2212.07677*, 2022.

Colin Wei, Yining Chen, and Tengyu Ma. Statistically meaningful approximation: a case study on approximating turing machines with transformers. *Advances in Neural Information Processing Systems*, 35:12071–12083, 2022a.

Jason Wei, Yi Tay, Rishi Bommasani, Colin Raffel, Barret Zoph, Sebastian Borgeaud, Dani Yogatama, Maarten Bosma, Denny Zhou, Donald Metzler, et al. Emergent abilities of large language models. *arXiv preprint arXiv:2206.07682*, 2022b.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Ed Chi, Quoc Le, and Denny Zhou. Chain of thought prompting elicits reasoning in large language models. *arXiv preprint arXiv:2201.11903*, 2022c.

Jerry Wei, Jason Wei, Yi Tay, Dustin Tran, Albert Webson, Yifeng Lu, Xinyun Chen, Hanxiao Liu, Da Huang, Denny Zhou, et al. Larger language models do in-context learning differently. *arXiv preprint arXiv:2303.03846*, 2023.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*, 2019.

Sang Michael Xie, Aditi Raghunathan, Percy Liang, and Tengyu Ma. An explanation of in-context learning as implicit bayesian inference. *arXiv preprint arXiv:2111.02080*, 2021.

Zheng Yuan, Hongyi Yuan, Chuanqi Tan, Wei Wang, and Songfang Huang. How well do large language models perform in arithmetic tasks? *arXiv preprint arXiv:2304.02015*, 2023.

Chulhee Yun, Srinadh Bhojanapalli, Ankit Singh Rawat, Sashank J Reddi, and Sanjiv Kumar. Are transformers universal approximators of sequence-to-sequence functions? *arXiv preprint arXiv:1912.10077*, 2019.

Xiao Zhang, Yaodong Yu, Lingxiao Wang, and Quanquan Gu. Learning one-hidden-layer relu networks via gradient descent. In *The 22nd international conference on artificial intelligence and statistics*, pages 1524–1534. PMLR, 2019.

Hattie Zhou, Azade Nova, Hugo Larochelle, Aaron Courville, Behnam Neyshabur, and Hanie Sedghi. Teaching algorithmic reasoning via in-context learning. *arXiv preprint arXiv:2211.09066*, 2022.

Appendix

Table of Contents

A Construction

A.1 The Transformer architecture

For the purpose of this proof, we consider encoder-based transformer architectures and assume that the positional encodings are appended to the input[3]. We also consider that the heads are added and each one has each own key, query and value weight matrices. Formally, we have

$$attn(\boldsymbol{X}) = \boldsymbol{X} + \sum_{h=1}^{H} \boldsymbol{W}_{\mathbf{V}}^{h}\boldsymbol{X}\, softmax((\boldsymbol{W}_{\mathbf{K}}^{h}\boldsymbol{X})^{\top}\boldsymbol{W}_{\mathbf{Q}}^{h}\boldsymbol{X}) \tag{A.1}$$

$$\texttt{TF}(\boldsymbol{X}) = attn(\boldsymbol{X}) + \boldsymbol{W}_2(\boldsymbol{W}_1 attn(\boldsymbol{X}) + \boldsymbol{b}_1\mathbf{1}_n)_+ + \boldsymbol{b}_2\mathbf{1}_n \tag{A.2}$$

where $\boldsymbol{X} \in \mathbb{R}^{d\times n}$, H is the number of heads used and $f(x) = (x)_+$ is the ReLU function. We also make use of the temperature λ, which is a parameter of the softmax. Specifically, $softmax(\boldsymbol{x}) = \{\mathrm{e}^{\lambda x_i}/\sum_j \mathrm{e}^{\lambda x_j}\}_i$. Notice that as $\lambda \to \infty$ we have $softmax(\boldsymbol{x}) \to \max_i x_i$. We also assume that the inputs are bounded; we denote with N_{max} the maximum sequence length of the model.

Assumption 2 *Each entry is bounded by some constant c, which implies that $\|\boldsymbol{X}\| \leq c'$, for some large c' that depends on the maximum sequence length and the width of the transformer.*

A.2 Positional encodings

In the constructions below we use a combination of the binary representation of each position as well as some additional information as described in the following sections. The binary representations as well the encodings we construct, require only logarithmic space with respect to the sequence length. Notice that binary representation of the positions satisfy the following two conditions:

[3]We note here that in terms of our construction adding the encodings or appending them to the input can be viewed in a similar manner. Specifically, we can consider that the up-projection step projects the input to zero padded vectors, while the encodings are orthogonal to that projection in the sense that the have zero in the non-zero coordinates of the input. In that case adding the positional encodings corresponds to appending them to the input.

1. Let $\mathbf{r}_i$ be the binary representation of the i-th position, then there exists $\varepsilon > 0$ such that $\mathbf{r}_i^\top \mathbf{r}_i > \mathbf{r}_i^\top \mathbf{r}_j + \varepsilon$ for all $j \neq i$.
2. There exists a one layer transformer that can implement the addition of two pointers (see Lemma 2).

One useful fact about the encodings is that as

A.3 Constructing some useful "Bblack-box" functions

We follow the construction of previous work [Giannou et al., 2023] and use the following individual implementations as they do. We repeat the statements here for convenience. The first lemma is similar to Akyürek et al. [2022].

Lemma 1 *A transformer with one layer, two heads, and embedding dimension of $\mathcal{O}(\log n + d)$, where d is the input dimension and n is the sequence length, can copy any block of the input to any other block of the input with error ε arbitrarily small.*

This is the move operation in Akyürek et al. [2022]. Notice that since Akyürek et al. [2022] use decoder only models they can only move things only from (i, j) to (i', j') such that $i' > i$ and $j' > j$. However, since we consider encoder models by repeating their proof we can get the result for arbitrary positions. We also consider the positional encodings of Giannou et al. [2023] which leads to the ε error.

Remark 2 *For the idea on how this is possible we illustrate a simple example. Assume that $N = 5$ and we want to copy and element from position (1,2) where the first element signifies the row and the second the column to position (3,5). The first step is to copy the second column to the fifth one; to do so we force the softmax matrix to acts as a permutation matrix,* i.e., $[\boldsymbol{e}_1\ \boldsymbol{e}_2\ \boldsymbol{e}_3\ \boldsymbol{e}_4\ \boldsymbol{e}_2]$ *where $\boldsymbol{e}_i$ is the one hot vector for the i-th row. This is achieved by the bianary representations by using as $\boldsymbol{W}_\mathbf{K}\boldsymbol{X} = [\mathbf{r}_1\ \ldots\ \mathbf{r}_5]$, $\boldsymbol{W}_\mathbf{Q}\boldsymbol{X} = [\mathbf{r}_1\ \ldots\ \mathbf{r}_2]$ and by using λ large enough the result follows. Then, we use a value matrix that only keeps the row 1 and permutes it to the row 3. Finally, to account for the residual, we use an extra head, which softmax is approximately the identity and the value weight matrix is zero expect for the first row which is again permuted to the third one. This description follows the construction of Akyürek et al. [2022].*

Lemma 2 *There exists a 1-hidden layer feedforward, ReLU network, with $8d$ activations in the hidden layer and d neurons in the output layer that when given two d-dimensional binary vectors representing two non-negative integers, can output the binary vector representation of their sum, as long as the sum is less than 2^{d+1}.*

Lemma 3 *Fix $\epsilon > 0$ and consider an input of the following form*

$$\boldsymbol{X} = \left[\begin{array}{c|c|c|ccc} \mathbf{A} & \mathbf{0} & \mathbf{0} & \ldots & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \ldots & \mathbf{0} \\ \mathbf{r}_{1:d} & \mathbf{r}_{1:d} & \mathbf{r}_{1:d} & \ldots & \mathbf{r}_{1:d} \\ \mathbf{R}'_1 & \mathbf{R}'_2 & \mathbf{R}'_3 & \ldots & \mathbf{R}'_d \end{array}\right].$$

where $\mathbf{A} \in \mathbb{R}^{d\times d}$; then there exists transformer-based function block with 4 layers, 1 head and dimensionality $r = 2d + 2\log d = O(d)$ that outputs the following matrix

$$\boldsymbol{X} = \left[\begin{array}{c|c|c|ccc} \mathbf{A}' & \mathbf{A}' & \mathbf{A}' & \ldots & \mathbf{A}' \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \ldots & \mathbf{0} \\ \mathbf{r}_{1:d} & \mathbf{r}_{1:d} & \mathbf{r}_{1:d} & \ldots & \mathbf{r}_{1:d} \\ \mathbf{R}'_1 & \mathbf{R}'_2 & \mathbf{R}'_3 & \ldots & \mathbf{R}'_d \end{array}\right].$$

where $\mathbf{A}' = \mathbf{A}^\top + \epsilon\mathbf{M}$, for some $\|\mathbf{M}\| \leq 1$. The error ϵ depends on the choice of the temperature λ, as it is a consequence of the `read/write` *operations.*

Lemma 4 *Let $\mathbf{A} \in \mathbb{R}^{d\times m}$ and $\mathbf{B} \in \mathbb{R}^{d\times n}$. Then for any $\epsilon > 0$ there exists a transformer-based function block with 2 layers, 1 head and width $r = O(d)$ that outputs the multiplication $\mathbf{A}^\top\mathbf{B} + \epsilon\mathbf{M}$, for some $\|\mathbf{M}\| \leq 1$.*

Remark 3 *Notice that based on the proof of this lemma, the matrices/scalars/vectors need to be in the same rows,* i.e., $\mathbf{Q} = [\mathbf{A} \quad \mathbf{B}]$. *By also appending the appropriate binary encodings we can move the output in any specific place we choose as in Lemma 1. Also, following the proof of the paper the input matrix is actually*

$$\boldsymbol{X} = \begin{bmatrix} \mathbf{Q} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{1}\mathbf{1}^\top & \mathbf{0} \\ \boldsymbol{I} & \mathbf{0} & \mathbf{0} \\ & \mathbf{r}^{(1)} & \\ & \mathbf{r}^{(2)} & \end{bmatrix}$$

where $\mathbf{r}^{(1)}, \mathbf{r}^{(2)}$ *are chosen as in Lemma 1 to specify the position of the result.*

A.4 Results on filtering

Lemma 5 *Assume that the input to a transformer layer is of the following form*

$$\boldsymbol{X} = \begin{bmatrix} \boldsymbol{x}_1 & \boldsymbol{x}_2 & \dots & \boldsymbol{x}_{n-1} & \boldsymbol{x}_n \\ \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} \\ b_1 & b_2 & \dots & b_{n-1} & b_n \\ b'_1 & b'_2 & \dots & b'_{n-1} & b_n \end{bmatrix} \tag{A.3}$$

where $b_i, b'_i \in \{0, 1\}$*, with zero indicating that the corresponding point should be ignored. Then there exists a transformer* TF *consisting only of a ReLU layer that performs this filtering, i.e.,*

$$\mathit{TF}(\boldsymbol{X}) = \begin{bmatrix} \mathbf{1}\{b_1 \neq 0\}\boldsymbol{x}_1 & \mathbf{1}\{b_2 \neq 0\}\boldsymbol{x}_2 & \dots & \mathbf{1}\{b_{n-1} \neq 0\}\boldsymbol{x}_{n-1} & \mathbf{1}\{b_n \neq 0\}\boldsymbol{x}_n \\ \mathbf{1}\{b'_1 \neq 0\}\boldsymbol{x}_1 & \mathbf{1}\{b'_2 \neq 0\}\boldsymbol{x}_2 & \dots & \mathbf{1}\{b'_{n-1} \neq 0\}\boldsymbol{x}_{n-1} & \mathbf{1}\{b'_n \neq 0\}\boldsymbol{x}_n \\ b_1 & b_2 & \dots & b_{n-1} & b_n \\ b'_1 & b'_2 & \dots & b'_{n-1} & b_n \end{bmatrix} \tag{A.4}$$

Proof. The layer is the following:

$$\mathtt{TF}(\boldsymbol{x}_i) = x_i + (-Cb_i - \boldsymbol{x}_i)_+ - (-Cb_i + \boldsymbol{x}_i)_+ \tag{A.5}$$

for some large constant C. Notice that if $b_i = 1$ the output is just $\boldsymbol{x}_i$. But if $b_i = 0$ then the output is zero. For the second set instead of using the bits b_i, we use the b'_i. ■

Remark 4 *Notice that if some* b_i *is instead of* 1, $1 \pm \varepsilon$, $\varepsilon < c/C$. *Then the output of the above layer would be*

$$\begin{aligned} \mathit{TF}(\boldsymbol{x}_i) &= \boldsymbol{x}_i + (-C \pm C\varepsilon - \boldsymbol{x}_i) - (-C \pm C\varepsilon + \boldsymbol{x}_i) && \text{(A.6)} \\ &= \boldsymbol{x}_i && \text{(A.7)} \end{aligned}$$

while if some $b_i = \pm\varepsilon$ *instead of zero, and assuming that* $\boldsymbol{x}_i > c > 0$ *or* $\boldsymbol{x}_i < -c < 0$ *the output would be*

$$\begin{aligned} \mathit{TF}(\boldsymbol{x}_i) &= \boldsymbol{x}_i + (-C\varepsilon - \boldsymbol{x}_i)_+ - (-Cb_i + \boldsymbol{x}_i)_+ && \text{(A.8)} \\ &= \boldsymbol{x}_i \pm C\varepsilon - \boldsymbol{x}_i && \text{(A.9)} \\ &\leq c && \text{(A.10)} \end{aligned}$$

If $|\boldsymbol{x}_i| \leq c$*, again the output would be less than or equal to* c*, where* c *can be arbitrarily small.*

But how we can create these bits for each different token we want to produce? We now describe for clarity how next word prediction is performed. An $d \times N$-dimensional input is given to the transformer architecture, which is up-projected using the embedding layer; afterwards the positional encodings are added/appended to the it. At the last layer, the last token predicted is appended to the input, while the rest of the tokens remain unchanged. The same holds for the actual input to the transformer after appending the encodings. The only difference of the new input is the $n+1$-th token. We consider encoder-based architectures in all of our lemmas below.

In the subsequent lemma we will try to construct and automated process that works along those guidelines. To do so we assume that the input to the transformer contains the following information:

1. An enumeration of the tokens from 1 to N.

2. The $\ln$ of the above enumeration.
3. Zeros for the tokens that correspond to the data points, 1 for each token that is the $\boldsymbol{x}_{\text{test}}$ or it is a prediction based on this.
4. An enumeration 1 to L for each one of the data points provided. For example, if we are given three sets of data we would have : $1 \dots L\ 1 \dots L\ 1 \dots L$.
5. Some extra information that is needed to implement a multiplication step as described in Lemma 4 and to move things to the correct place.

The above information can be viewed as part of the encodings that are appended to the transformer. Formally, we have

Lemma 6 *Consider that a prompt with n in-context samples is given and the $\ell-1$-th prediction has been made, and the transformer is to predict the ℓ-th one. Assume the input to the transformer is:*

$$\boldsymbol{X} = \left[\begin{array}{ccccccccccccc}
\boldsymbol{x}_1 & \dots & \boldsymbol{s}_1^{\ell-1} & \boldsymbol{s}_1^{\ell} & \dots & \boldsymbol{s}_2^{\ell-1} & \boldsymbol{s}_2^{\ell} & \dots & \boldsymbol{s}_n^{L} & \boldsymbol{x}_{test} & \dots & \hat{\boldsymbol{s}}^{\ell-1} \\
\mathbf{0} & \dots & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} \\
\hline
1 & \dots & \ell & \ell+1 & \dots & \ell & \ell+1 & \dots & L+1 & 1 & \dots & \ell \\
1 & \dots & \ell & \ell+1 & \dots & \ell & \ell+1 & \dots & L+1 & 1 & \dots & \ell \\
\hline
1 & \dots & \ell & \ell+1 & \dots & L+\ell+1 & L+\ell+2 & \dots & n(L+1) & n(L+1)+1 & \dots & N \\
\ln(1) & \dots & \ln(\ell) & \ln(\ell+1) & \dots & \ln(L+\ell+1) & \ln(L+\ell+2) & \dots & \ln(n(L+1)) & \ln(n(L+1)+1) & \dots & \ln(N) \\
0 & \dots & 0 & 0 & \dots & 0 & 0 & \dots & 0 & 1 & \dots & 1 \\
1 & \dots & 1 & 1 & \dots & 1 & 1 & \dots & 1 & 1 & \dots & 1 \\
\boldsymbol{m}_1 & \dots & \boldsymbol{m}_\ell & \boldsymbol{m}_{\ell+1} & \dots & \boldsymbol{m}_{L+\ell+1} & \boldsymbol{m}_{L+\ell+2} & \dots & \boldsymbol{m}_{n(L+1)} & \boldsymbol{m}_{n(L+1)+1} & \dots & \boldsymbol{m}_N
\end{array}\right]$$

where $(\hat{\boldsymbol{s}}^i)_{i=1}^{\ell-1}$ denote the first $\ell-1$ recurrent outputs of TF and for simplicity, let $N := n(L+1)+\ell$ denote the total number of tokens. Then there exists a transformer TF consisting of 4 layers that has as output

$$\texttt{TF}(\boldsymbol{X}) = \left[\begin{array}{ccccccccccccc}
\mathbf{0} & \dots & \boldsymbol{s}_1^{\ell-1} & \mathbf{0} & \dots & \boldsymbol{s}_2^{\ell-1} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} & \dots & \hat{\boldsymbol{s}}^{\ell-1} \\
\mathbf{0} & \dots & \mathbf{0} & \boldsymbol{s}_1^{\ell} & \dots & \mathbf{0} & \boldsymbol{s}_2^{\ell} & \dots & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} \\
\hline
0 & \dots & 1 & 0 & \dots & 1 & 0 & \dots & 0 & 0 & \dots & 1 \\
0 & \dots & 0 & 1 & \dots & 0 & 1 & \dots & 0 & 0 & \dots & 0 \\
\hline
1 & \dots & \ell & \ell+1 & \dots & L+\ell+1 & L+\ell+2 & \dots & n(L+1) & n(L+1)+1 & \dots & N \\
\ln(1) & \dots & \ln(\ell) & \ln(\ell+1) & \dots & \ln(L+\ell+1) & \ln(L+\ell+2) & \dots & \ln(n(L+1)) & \ln(n(L+1)+1) & \dots & \ln(N) \\
0 & \dots & 0 & 0 & \dots & 0 & 0 & \dots & 0 & 1 & \dots & 1 \\
1 & \dots & 1 & 1 & \dots & 1 & 1 & \dots & 1 & 1 & \dots & 1 \\
\boldsymbol{m}_1 & \dots & \boldsymbol{m}_\ell & \boldsymbol{m}_{\ell+1} & \dots & \boldsymbol{m}_{L+\ell+1} & \boldsymbol{m}_{L+\ell+2} & \dots & \boldsymbol{m}_{n(L+1)} & \boldsymbol{m}_{n(L+1)+1} & \dots & \boldsymbol{m}_N
\end{array}\right]$$

with error up to $\delta\mathbf{M}$, where $\|\mathbf{M}\| \le 1$ and $\delta > 0$ is a constant that is controlled and can be arbitrarily small.

Proof. Since the error of each step we use is controlled and can be arbitrarily small, we can omit it here in the sequence to make the proof simpler. Each one of the seven layers will induce an error of the form $\varepsilon_i \mathbf{M}_i$ with $\|\mathbf{M}_i\| \le 1$; at the end the error is aggregated and we have a total error of $\sum_{i=1}^{7} \varepsilon_i \mathbf{M}_i$ with $\|\sum_{i=1}^{7} \varepsilon_i \mathbf{M}_i\| \le \sum_{i=1}^{7} \varepsilon_i$. By setting $\varepsilon_i = \delta/7$ we get the desired error bound.

Step 1: Extract the sequence length (1 layer). Let

$$\boldsymbol{W}_{\mathbf{K}} = [\mathbf{0}\ \ \mathbf{0}\ \ 0\ \ 0\ \ 0\ \ 1\ \ 0\ \ 0\ \ \mathbf{0}] \quad \boldsymbol{W}_{\mathbf{Q}} = [\mathbf{0}\ \ \mathbf{0}\ \ 0\ \ 0\ \ 0\ \ 0\ \ 0\ \ 0\ \ 1\ \ \mathbf{0}] \tag{A.11}$$

and thus

$$(\boldsymbol{W}_{\mathbf{K}}\boldsymbol{X})^\top (\boldsymbol{W}_{\mathbf{Q}}\boldsymbol{X}) = \begin{bmatrix} \ln(1) & \ln(1) & \dots & \ln(1) \\ \ln(2) & \ln(2) & \dots & \ln(2) \\ \vdots & \vdots & \ddots & \vdots \\ \ln(\mathrm{N}) & \ln(\mathrm{N}) & \dots & \ln(\mathrm{N}) \end{bmatrix}. \tag{A.12}$$

So after the softmax is applied we have

$$
\textit{softmax}((\boldsymbol{W}_{\mathbf{K}}\boldsymbol{X})^\top(\boldsymbol{W}_{\mathbf{Q}}\boldsymbol{X})) = \begin{bmatrix} \frac{1}{\sum_{i=1}^{\mathrm{N}} i} & \frac{1}{\sum_{i=1}^{\mathrm{N}} i} & \cdots & \frac{1}{\sum_{i=1}^{\mathrm{N}} i} \\ \frac{2}{\sum_{i=1}^{\mathrm{N}} i} & \frac{2}{\sum_{i=1}^{\mathrm{N}} i} & \cdots & \frac{2}{\sum_{i=1}^{\mathrm{N}} i} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\mathrm{N}}{\sum_{i=1}^{\mathrm{N}} i} & \frac{\mathrm{N}}{\sum_{i=1}^{\mathrm{N}} i} & \cdots & \frac{\mathrm{N}}{\sum_{i=1}^{\mathrm{N}} i} \end{bmatrix}. \tag{A.13}
$$

We then set the weight value matrix as to zero-out all lines except for one line as follows

$$
\boldsymbol{W}_{\mathbf{V}}\boldsymbol{X} = \begin{bmatrix} \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} \\ 1 & 2 & \dots & \mathrm{N} \\ \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} \end{bmatrix}. \tag{A.14}
$$

After adding the residual and using an extra head where the softmax returns identity matrix and the value weight matrix is minus the identity, we get attenton output

$$
\left[\begin{array}{ccccccccccccc}
\boldsymbol{x}_1 & \dots & \boldsymbol{s}_1^{\ell-1} & \boldsymbol{s}_1^{\ell} & \dots & \boldsymbol{s}_2^{\ell-1} & \boldsymbol{s}_2^{\ell} & \dots & \boldsymbol{s}_n^{L} & \boldsymbol{x}_{\text{test}} & \dots & \hat{\boldsymbol{s}}^{\ell-1} \\
\mathbf{0} & \dots & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} \\
\hline
1 & \dots & \ell & \ell+1 & \dots & \ell & \ell+1 & \dots & L+1 & 1 & \dots & \ell \\
1 & \dots & \ell & \ell+1 & \dots & \ell & \ell+1 & \dots & L+1 & 1 & \dots & \ell \\
\hline
\frac{\sum_{i=1}^{\mathrm{N}} i^2}{\sum_{i=1}^{\mathrm{N}} i} & \dots & \frac{\sum_{i=1}^{\mathrm{N}} i^2}{\sum_{i=1}^{\mathrm{N}} i} & \frac{\sum_{i=1}^{\mathrm{N}} i^2}{\sum_{i=1}^{\mathrm{N}} i} & \dots & \frac{\sum_{i=1}^{\mathrm{N}} i^2}{\sum_{i=1}^{\mathrm{N}} i} & \frac{\sum_{i=1}^{\mathrm{N}} i^2}{\sum_{i=1}^{\mathrm{N}} i} & \dots & \frac{\sum_{i=1}^{\mathrm{N}} i^2}{\sum_{i=1}^{\mathrm{N}} i} & \frac{\sum_{i=1}^{\mathrm{N}} i^2}{\sum_{i=1}^{\mathrm{N}} i} & \dots & \frac{\sum_{i=1}^{\mathrm{N}} i^2}{\sum_{i=1}^{\mathrm{N}} i} \\
\ln(1) & \dots & \ln(\ell) & \ln(\ell+1) & \dots & \ln(L+\ell+1) & \ln(L+\ell+2) & \dots & \ln(n(L+1)) & \ln(n(L+1)+1) & \dots & \ln(\mathrm{N}) \\
0 & \dots & 0 & 0 & \dots & 0 & 0 & \dots & 0 & 1 & \dots & 1 \\
1 & \dots & 1 & 1 & \dots & 1 & 1 & \dots & 1 & 1 & \dots & 1 \\
\boldsymbol{m}_1 & \dots & \boldsymbol{m}_{\ell} & \boldsymbol{m}_{\ell+1} & \dots & \boldsymbol{m}_{L+\ell+1} & \boldsymbol{m}_{L+\ell+2} & \dots & \boldsymbol{m}_{n(L+1)} & \boldsymbol{m}_{n(L+1)+1} & \dots & \boldsymbol{m}_{\mathrm{N}}
\end{array}\right].
$$

Notice that $\frac{\sum_{i=1}^{\mathrm{N}} i^2}{\sum_{i=1}^{\mathrm{N}} i} = \frac{2\mathrm{N}(\mathrm{N}+1)(2\mathrm{N}+1)}{6\mathrm{N}(\mathrm{N}+1)} = \frac{2\mathrm{N}+1}{3}$. We then use the ReLU layer as to multiply with $3/2$ and subtract 1 from this column. This results to attention output

$$
\left[\begin{array}{ccccccccccccc}
\boldsymbol{x}_1 & \dots & \boldsymbol{s}_1^{\ell-1} & \boldsymbol{s}_1^{\ell} & \dots & \boldsymbol{s}_2^{\ell-1} & \boldsymbol{s}_2^{\ell} & \dots & \boldsymbol{s}_n^{L} & \boldsymbol{x}_{\text{test}} & \dots & \hat{\boldsymbol{s}}^{\ell-1} \\
\mathbf{0} & \dots & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} \\
\hline
1 & \dots & \ell & \ell+1 & \dots & \ell & \ell+1 & \dots & L+1 & 1 & \dots & \ell \\
1 & \dots & \ell & \ell+1 & \dots & \ell & \ell+1 & \dots & L+1 & 1 & \dots & \ell \\
\hline
\mathrm{N} & \dots & \mathrm{N} & \mathrm{N} & \dots & \mathrm{N} & \mathrm{N} & \dots & \mathrm{N} & \mathrm{N} & \dots & \mathrm{N} \\
\ln(1) & \dots & \ln(\ell) & \ln(\ell+1) & \dots & \ln(L+\ell+1) & \ln(L+\ell+2) & \dots & \ln(n(L+1)) & \ln(n(L+1)+1) & \dots & \ln(\mathrm{N}) \\
0 & \dots & 0 & 0 & \dots & 0 & 0 & \dots & 0 & 1 & \dots & 1 \\
1 & \dots & 1 & 1 & \dots & 1 & 1 & \dots & 1 & 1 & \dots & 1 \\
\boldsymbol{m}_1 & \dots & \boldsymbol{m}_{\ell} & \boldsymbol{m}_{\ell+1} & \dots & \boldsymbol{m}_{L+\ell+1} & \boldsymbol{m}_{L+\ell+2} & \dots & \boldsymbol{m}_{n(L+1)} & \boldsymbol{m}_{n(L+1)+1} & \dots & \boldsymbol{m}_{\mathrm{N}}
\end{array}\right]
$$

Step 2: Extract the identifier of the prediction to be made ℓ (3 layers). Now, by setting the key and query weight matrices of the attention as to just keep the all ones row, we get a matrix that attends equally to all tokens. Then the value weight matrix keeps the row that contains ℓ ones and thus we

get the number $\frac{\ell}{\mathrm{N}}$, propagated in all the sequence length. Then the attention output is as follows:

$$\begin{bmatrix}
\boldsymbol{x}_1 & \dots & \boldsymbol{s}_1^{\ell-1} & \boldsymbol{s}_1^{\ell} & \dots & \boldsymbol{s}_2^{\ell-1} & \boldsymbol{s}_2^{\ell} & \dots & \boldsymbol{s}_n^{L} & \boldsymbol{x}_{\text{test}} & \dots & \hat{\boldsymbol{s}}^{\ell-1} \\
\mathbf{0} & \dots & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} \\
\hline
1 & \dots & \ell & \ell+1 & \dots & \ell & \ell+1 & \dots & L+1 & 1 & \dots & \ell \\
1 & \dots & \ell & \ell+1 & \dots & \ell & \ell+1 & \dots & L+1 & 1 & \dots & \ell \\
\hline
\mathrm{N} & \dots & \mathrm{N} & \mathrm{N} & \dots & \mathrm{N} & \mathrm{N} & \dots & \mathrm{N} & \mathrm{N} & \dots & \mathrm{N} \\
\frac{\ell}{\mathrm{N}} & \dots & \frac{\ell}{\mathrm{N}} & \frac{\ell}{\mathrm{N}} & \dots & \frac{\ell}{\mathrm{N}} & \frac{\ell}{\mathrm{N}} & \dots & \frac{\ell}{\mathrm{N}} & \frac{\ell}{\mathrm{N}} & \dots & \frac{\ell}{\mathrm{N}} \\
0 & \dots & 0 & 0 & \dots & 0 & 0 & \dots & 0 & 1 & \dots & 1 \\
1 & \dots & 1 & 1 & \dots & 1 & 1 & \dots & 1 & 1 & \dots & 1 \\
\boldsymbol{m}_1 & \dots & \boldsymbol{m}_\ell & \boldsymbol{m}_{\ell+1} & \dots & \boldsymbol{m}_{L+\ell+1} & \boldsymbol{m}_{L+\ell+2} & \dots & \boldsymbol{m}_{n(L+1)} & \boldsymbol{m}_{n(L+1)+1} & \dots & \boldsymbol{m}_{\mathrm{N}}
\end{bmatrix}. \tag{A.15}$$

As described in Lemma 4 to implement multiplication of two values up to any error ϵ, we need 1) have the two numbers to be multiplied next to each other and an extra structure (some constants, which we consider that are encoded in the last rows of the matrix $\boldsymbol{m}_i$) and 2) the corresponding structure presented in Lemma 4 . So we need to make the following transformation in the rows containing N and $\frac{\ell}{\mathrm{N}}$

$$\begin{bmatrix} \mathrm{N} & \mathrm{N} & \dots & \mathrm{N} \\ \frac{\ell}{\mathrm{N}} & \frac{\ell}{\mathrm{N}} & \dots & \frac{\ell}{\mathrm{N}} \end{bmatrix} \rightarrow \begin{bmatrix} \mathrm{N} & \frac{\ell}{\mathrm{N}} & \dots & \mathrm{N} \\ \frac{\ell}{\mathrm{N}} & \frac{\ell}{\mathrm{N}} & \dots & \frac{\ell}{\mathrm{N}} \end{bmatrix} \tag{A.16}$$

$$\rightarrow \begin{bmatrix} \ell & * & \dots & * \\ * & * & \dots & * \end{bmatrix} \tag{A.17}$$

$$\rightarrow \begin{bmatrix} \ell & \ell & \dots & \ell \\ * & * & \dots & * \end{bmatrix} \tag{A.18}$$

where $*$ denotes inconsequential values. For the first step we assume that we have the necessary binary representations [4] and then we use Lemma 1 which shows how we can perform the operation of copy/paste. For the second step now we use Lemma 4 to perform the multiplication, this will affect some of the other values. Then for the last step consider the follow (sub-)rows of the matrix $\boldsymbol{X}$

$$\begin{bmatrix} \ell & * & * & \dots & * \\ \mathbf{r}_1 & \mathbf{r}_1 & \mathbf{r}_1 & \dots & \mathbf{r}_1 \\ \mathbf{r}_1 & \mathbf{r}_2 & \mathbf{r}_3 & \dots & \mathbf{r}_{\mathrm{N}} \end{bmatrix} \tag{A.19}$$

where $\mathbf{r}_i$ is the binary representation of position i. By choosing $\boldsymbol{W}_{\mathbf{K}}, \boldsymbol{W}_{\mathbf{Q}}$ as to

$$\boldsymbol{W}_{\mathbf{K}}\boldsymbol{X} = \begin{bmatrix} \mathbf{r}_1 & \mathbf{r}_2 & \dots & \mathbf{r}_{\mathrm{N}} \end{bmatrix}, \quad \boldsymbol{W}_{\mathbf{Q}}\boldsymbol{X} = \begin{bmatrix} \mathbf{r}_1 & \mathbf{r}_1 & \dots & \mathbf{r}_1 \end{bmatrix} \tag{A.20}$$

[4] the size that we need will be $2 \log \mathrm{N}_{\text{max}} + 1$, where N_{max} is the maximum sequence length

and consider $\boldsymbol{W_V X} = \begin{bmatrix} \ell & * & \dots & * \\ \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} \end{bmatrix}$ we have that

$$
\begin{aligned}
\textit{attn}(\boldsymbol{X}) &= \boldsymbol{X} + \boldsymbol{W_V X}\, \textit{softmax}((\boldsymbol{W_K X})^\top \boldsymbol{W_Q X}) \\
&= \boldsymbol{X} + \begin{bmatrix} \ell & * & \dots & * \\ \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} \end{bmatrix} \textit{softmax}\left(\begin{bmatrix} \mathbf{r}_1^\top \mathbf{r}_1 & \mathbf{r}_1^\top \mathbf{r}_1 & \dots & \mathbf{r}_1^\top \mathbf{r}_1 \\ \mathbf{r}_1^\top \mathbf{r}_2 & \mathbf{r}_1^\top \mathbf{r}_2 & \dots & \mathbf{r}_1^\top \mathbf{r}_2 \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{r}_1^\top \mathbf{r}_\mathrm{N} & \mathbf{r}_1^\top \mathbf{r}_\mathrm{N} & \dots & \mathbf{r}_1^\top \mathbf{r}_\mathrm{N} \end{bmatrix} \right) \\
&= \boldsymbol{X} + \begin{bmatrix} \ell & * & \dots & * \\ \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} \end{bmatrix} \begin{bmatrix} 1 & 1 & \dots & 1 \\ 0 & 0 & \dots & 0 \\ \vdots & \vdots & \dots & \vdots \\ 0 & 0 & \dots & 0 \end{bmatrix} + \varepsilon \mathbf{M}
\end{aligned}
$$

By subtracting one identity head for the first that we focus on as described in Lemma 1 we have that $\textit{attn}(\boldsymbol{X})$ results in the desired matrix. We output this result and overwrite ℓ/N, thus we have

$$
\boldsymbol{X} = \left[\begin{array}{cccccccccccc}
\boldsymbol{x}_1 & \dots & \boldsymbol{s}_1^{\ell-1} & \boldsymbol{s}_1^{\ell} & \dots & \boldsymbol{s}_2^{\ell-1} & \boldsymbol{s}_2^{\ell} & \dots & \boldsymbol{s}_n^{L} & \boldsymbol{x}_{\text{test}} & \dots & \hat{\boldsymbol{s}}^{\ell-1} \\
\mathbf{0} & \dots & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} \\
\hline
1 & \dots & \ell & \ell+1 & \dots & \ell & \ell+1 & \dots & L+1 & 1 & \dots & \ell \\
1 & \dots & \ell & \ell+1 & \dots & \ell & \ell+1 & \dots & L+1 & 1 & \dots & \ell \\
\hline
\mathrm{N} & \dots & \mathrm{N} & \mathrm{N} & \dots & \mathrm{N} & \mathrm{N} & \dots & \mathrm{N} & \mathrm{N} & \dots & \mathrm{N} \\
\ell & \dots & \ell & \ell & \dots & \ell & \ell & \dots & \ell & \ell & \dots & \ell \\
0 & \dots & 0 & 0 & \dots & 0 & 0 & \dots & 0 & 1 & \dots & 1 \\
1 & \dots & 1 & 1 & \dots & 1 & 1 & \dots & 1 & 1 & \dots & 1 \\
\boldsymbol{m}_1 & \dots & \boldsymbol{m}_\ell & \boldsymbol{m}_{\ell+1} & \dots & \boldsymbol{m}_{L+\ell+1} & \boldsymbol{m}_{L+\ell+2} & \dots & \boldsymbol{m}_{n(L+1)} & \boldsymbol{m}_{n(L+1)+1} & \dots & \boldsymbol{m}_\mathrm{N}
\end{array}\right] \tag{A.21}
$$

Step 3: Create $\ell+1$ (0 layer). We first copy the row with ℓ to the row with N, this can trivially be done with a ReLU layer that outputs zero everywhere else except for the row of Ns that output $(\ell)_+ - (N)_+$ to account also for the residual. We now use one of the bias terms (notice that ℓ is always positive) and set to one in one of the two rows that contain the ℓ, again we account for the residual as before; everything else remains unchanged. Thus, we have

$$
\boldsymbol{X} = \left[\begin{array}{cccccccccccc}
\boldsymbol{x}_1 & \dots & \boldsymbol{s}_1^{\ell-1} & \boldsymbol{s}_1^{\ell} & \dots & \boldsymbol{s}_2^{\ell-1} & \boldsymbol{s}_2^{\ell} & \dots & \boldsymbol{s}_n^{L} & \boldsymbol{x}_{\text{test}} & \dots & \hat{\boldsymbol{s}}^{\ell-1} \\
\mathbf{0} & \dots & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} & \mathbf{0} & \dots & \mathbf{0} \\
\hline
1 & \dots & \ell & \ell+1 & \dots & \ell & \ell+1 & \dots & L+1 & 1 & \dots & \ell \\
1 & \dots & \ell & \ell+1 & \dots & \ell & \ell+1 & \dots & L+1 & 1 & \dots & \ell \\
\hline
\ell+1 & \dots & \ell+1 & \ell+1 & \dots & \ell+1 & \ell+1 & \dots & \ell+1 & \ell+1 & \dots & \ell+1 \\
\ell & \dots & \ell & \ell & \dots & \ell & \ell & \dots & \ell & \ell & \dots & \ell \\
0 & \dots & 0 & 0 & \dots & 0 & 0 & \dots & 0 & 1 & \dots & 1 \\
1 & \dots & 1 & 1 & \dots & 1 & 1 & \dots & 1 & 1 & \dots & 1 \\
\boldsymbol{m}_1 & \dots & \boldsymbol{m}_\ell & \boldsymbol{m}_{\ell+1} & \dots & \boldsymbol{m}_{L+\ell+1} & \boldsymbol{m}_{L+\ell+2} & \dots & \boldsymbol{m}_{n(L+1)} & \boldsymbol{m}_{n(L+1)+1} & \dots & \boldsymbol{m}_\mathrm{N}
\end{array}\right] \tag{A.22}
$$

This operation can collectively be implemented (add the bias + copy the row) in the ReLU layer of the previous transformer layer that was not used in the previous step.

Step 4: Create the binary bits (2 layers). We will now use the information extracted in the previous steps, to create the binary indicators/bit to identify which tokens we want to filter. This can be easily implemented with one layer of transformer and especially the ReLU part of it. Notice that if we subtract the row that contains the tokens that have already been predicted, *i.e.,* $[\ell\ \ell \dots\ \ell]$ from the row that contains $[1\ 2 \dots L\ 1\ 2 \dots L \dots L]$ we will get zero only in the positions that we want to filter out and some non-zero quantity in the rest. This is trivially implemented with one ReLU layer. So, we need to implement an if..then type of operation. Basically, if the quantity at hand is zero we want to set the bit to one, while if it non-zero to set it to be zero. This can be implemented with the following ReLU part of a transformer layer

$$\texttt{TF}(x_i) = 1 - (x_i)_+ - (-x_i)_+ + (x_i - 1)_+ + (-x_i - 1)_+ - ((x_i)_+ - (-x_i)_+) \tag{A.23}$$

the last two terms are to account for the residual. Again the rest of the rows do not change and are zeroed-out.

Step 5: Implement the filtering (1 layer). We now apply Lemma 5 and our proof is completed. ■

Theorem 2 (Theorem 1 restated) *Consider a prompt $\boldsymbol{p}_n(f)$ generated from an L-layer MLP $f(\cdot)$ as described in Definition 1, and assume given test example $(\boldsymbol{x}_{test}, \boldsymbol{s}^1_{test}, \dots \boldsymbol{s}^L_{test})$. For any resolution $\epsilon > 0$, there exists $\delta = \delta(\epsilon)$, iteration choice $T = \mathcal{O}(\kappa^2_{\max} \log(1/\epsilon))$, and a backend transformer construction TF_{BE} such that the concatenated transformer $TF = TF_{LR} \circ TF_{BE}$ implements the following: Let $(\hat{\boldsymbol{s}}^i)_{i=1}^{\ell-1}$ denote the first $\ell - 1$ CoT-I/O outputs of TF and set $\boldsymbol{p}[\ell] = (\boldsymbol{p}_n(f), \boldsymbol{x}_{test}, \hat{\boldsymbol{s}}^1 \dots \hat{\boldsymbol{s}}^{\ell-1})$. At step ℓ, TF implements*

1. ***Filtering.*** *Define the filtered prompt with input/output features of layer ℓ,*

$$\boldsymbol{p}_n^{filter} = \begin{pmatrix} \dots \mathbf{0}, \boldsymbol{s}_1^{\ell-1}, \mathbf{0} \dots \mathbf{0}, \boldsymbol{s}_n^{\ell-1}, \mathbf{0} \dots \mathbf{0}, \hat{\boldsymbol{s}}^{\ell-1} \\ \dots \mathbf{0}, \mathbf{0}, \boldsymbol{s}_1^{\ell} \dots \mathbf{0}, \mathbf{0}, \boldsymbol{s}_n^{\ell} \dots \mathbf{0}, \mathbf{0} \end{pmatrix}.$$

   There exists a fixed projection matrix $\boldsymbol{\Pi}$ that applies individually on tokens such that the backend output obeys $\|\boldsymbol{\Pi}(TF_{BE}(\boldsymbol{p}[\ell])) - \boldsymbol{p}_n^{filter}\| \le \delta$.

2. ***Gradient descent.*** *The combined model obeys $\|TF(\boldsymbol{p}[\ell]) - \boldsymbol{s}^{\ell}_{test}\| \le \ell \cdot \epsilon / L$.*

TF_{BE} has constant number of layers independent of T and n. Consequently, after L rounds of CoT-I/O, TF outputs $f(\boldsymbol{x}_{test})$ up to ϵ accuracy.

Proof. We apply Lemma 6 from which it is clear that there exists a projection such that the result stated in (1. Filtering) holds, and it is independent to T, n and ℓ. Next we turn to prove (2. Gradient descent). Since in Definition 1, we assume that the network's activation function is leaky-ReLU, *i.e.,*

$$\phi(x) = \begin{cases} x, & \text{if } x \ge 0 \\ \alpha x, & \text{otherwise.} \end{cases} \tag{A.24}$$

Thus, as a first step we construct the inverse of leaky-ReLU and apply it in the second row of $\boldsymbol{p}_n^{\text{filter}}$ where the inverse of leaky-ReLU is

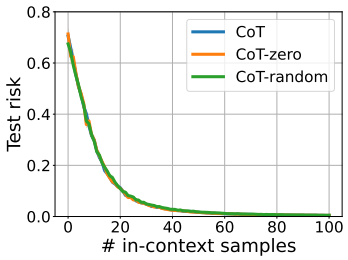
$$\phi^{-1}(y) = \begin{cases} y, & \text{if } y \ge 0 \\ y/\alpha, & \text{otherwise.} \end{cases} \tag{A.25}$$

This can be implemented with the following activation function (denoted by $\sigma(\cdot)$) using ReLU:

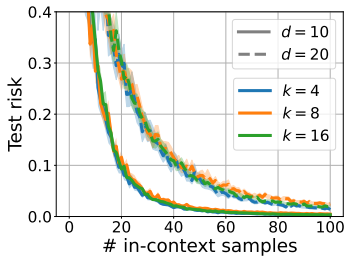
$$\sigma(x) = (x)_+ - 1/\alpha(-x)_+. \tag{A.26}$$

After it, it remains $\texttt{TF}_{\texttt{LR}}$ to solve linear regression problems. Taking ℓth layer, first neuron prediction as an example, and letting $\boldsymbol{x}'_i := \boldsymbol{s}_i^{\ell-1}$, $y'_i := \phi^{-1}(\boldsymbol{s}_i^{\ell}[0])$ and $\boldsymbol{w} = \boldsymbol{W}_\ell[0]$, linear regression has form of $y'_i = \boldsymbol{w}^\top \boldsymbol{x}'_i$ for $i \in [n]$. Notice that the extra zeros do not contribute in the update performed by gradient descent, and after gradient descent has been performed, we apply back the leaky ReLU. Then following Assumption 1, since we assume $\texttt{TF}_{\texttt{LR}}$ performs the same as gradient descent optimizer, given matrix condition as described in Definition 1, after running T iterations of gradient descend on the linear regression problem and considering each layer prediction with resolution ϵ/L, we can get that $\|\texttt{TF}_{\texttt{LR}}(\boldsymbol{p}[\ell]) - \bar{\boldsymbol{s}}^{\ell}\| \le \epsilon/L$, where $\bar{\boldsymbol{s}}^{\ell} = \phi(\boldsymbol{W}_\ell \hat{\boldsymbol{s}}^{\ell-1})$ is the correct prediction if taking $\hat{\boldsymbol{s}}^{\ell-1}$ as input. Then we have

$$\|\texttt{TF}_{\texttt{LR}}(\boldsymbol{p}[\ell] - \boldsymbol{s}^{\ell}_{\text{test}})\| \le \|\texttt{TF}_{\texttt{LR}}(\boldsymbol{p}[\ell]) - \bar{\boldsymbol{s}}^{\ell}\| + \|\boldsymbol{W}_\ell(\hat{\boldsymbol{s}}^{\ell-1} - \boldsymbol{s}^{\ell-1}_{\text{test}})\| \lesssim \epsilon/L + \|\texttt{TF}_{\texttt{LR}}(\boldsymbol{p}[\ell-1] - \boldsymbol{s}^{\ell-1}_{\text{test}})\|.$$



(a) Test filtering on first layer prediction



(b) CoT-I/O: composed risk (same as Fig. 2(a))

Figure 8: Fig. 8(a) presents a filtering evidence of 2-layer MLPs. Given a 2-layer MLP in-context example $(\boldsymbol{x}, \boldsymbol{s}, y)$, CoT admits $(\boldsymbol{x}, \boldsymbol{s}, y)$ as test sample; while test samples of CoT-zero and CoT-random are formed by $(\boldsymbol{x}, \boldsymbol{s}, 0)$ and $(\boldsymbol{x}, \boldsymbol{s}, z)$ where $z \sim \mathcal{N}(0, d)$. Fig. 8(b) is directly cloned form Fig. 2(a) with error bar for better comparison with results in Fig. 9.

Let $\texttt{TF}_{\texttt{LR}}(\boldsymbol{p}[0])$ returns $\boldsymbol{x}_{\text{test}}$ and therefore $\|\texttt{TF}_{\texttt{LR}}(\boldsymbol{p}[0]) - \boldsymbol{x}_{\text{test}}\| = 0$. Combing results in that $\|\texttt{TF}_{\texttt{LR}}(\boldsymbol{p}[\ell] - \boldsymbol{s}^{\ell}_{\text{test}})\| \lesssim \ell \cdot \epsilon/L$. Since from Lemma 6 we have that we can choose δ to be arbitrary. Let $\delta = \epsilon/L$, where L is the total predictions we will make. Then it will result in $\|\texttt{TF}(\boldsymbol{p}[\ell] - \boldsymbol{s}^{\ell}_{\text{test}})\| \lesssim \ell \cdot \epsilon/L$, which completes the proof.

■

B Experimental Details

In this section, we provide the implementation details of our experiments.

B.1 Model evaluation

Recap the same setting as in Section 4.1 and assume we have pretrained models with parameters $\hat{\boldsymbol{\theta}}^{\text{CoT-I}}$ and $\hat{\boldsymbol{\theta}}^{\text{CoT-I/O}}$. Next we make predictions following Section 2.2. Letting $\ell(\cdot, \cdot) : \mathcal{Y} \times \mathcal{Y} \to \mathbb{R}$ be loss function, we can define test risks as follows.

$$\mathcal{L}^{\text{CoT-I}}(n) = \mathbb{E}_{(\boldsymbol{x}_i)_{i=1}^{n}, (f_\ell)_{\ell=1}^{L}} \left[\ell(\hat{\boldsymbol{y}}_n, f(\boldsymbol{x}_n))\right] \quad \text{where} \quad \hat{\boldsymbol{y}}_n = \texttt{TF}(\boldsymbol{p}_n(f), \boldsymbol{x}_n; \hat{\boldsymbol{\theta}}^{\text{CoT-I}})$$

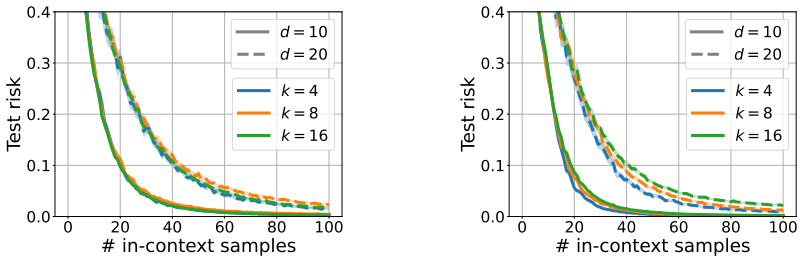
and

$$\mathcal{L}^{\text{CoT-I/O}}(n) = \mathbb{E}_{(\boldsymbol{x}_i)_{i=1}^{n}, (f_\ell)_{\ell=1}^{L}} \left[\ell(\hat{\boldsymbol{y}}_n, f(\boldsymbol{x}_n))\right] \quad \text{where} \quad \hat{\boldsymbol{y}}_n = \texttt{TF}(\boldsymbol{p}_n(f), \boldsymbol{x}_n, \hat{\boldsymbol{s}}^1 \cdots, \hat{\boldsymbol{s}}^{L-1}; \hat{\boldsymbol{\theta}}^{\text{CoT-I/O}}).$$

Here, we use $\mathcal{L}(n)$ to define the test risk when given prompt with n in-context samples. Then, results shown in Figures 2&3&4&5&6(a) are test risks $\mathcal{L}(n)$ given $n \in [N]$. Following model training and evaluation, we can see that once loss function is the same for both training and predicting, CoT-I (as well as ICL) accepts training risk $\mathcal{L}^{\text{CoT-I}}_{\text{train}} = \frac{1}{N}\sum_{n=1}^{N} \mathcal{L}^{\text{CoT-I}}(n)$.

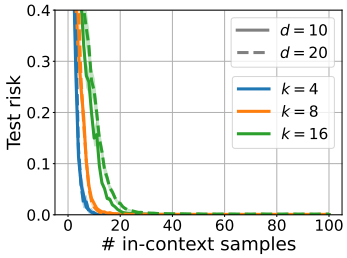
B.2 Implementation

All the transformer experiments use the GPT-2 model [Radford et al., 2019] and our codebase is based on prior works [Garg et al., 2022, Wolf et al., 2019]. Specifically, the model is trained using the `Adam` optimizer with learning rate 0.0001 and batch size 64, and we train 500k iterations in total for all ICL, CoT-I and CoT-I/O methods. Each iteration randomly samples inputs $\boldsymbol{x}$s and functions fs. We also apply curriculum learning over the prompt n as Garg et al. [2022] did for 2-layer random MLPs (Sec. 4.2). For both training and testing, we use the squared error as the loss function, *i.e.*, $\ell(\hat{\boldsymbol{y}}, \boldsymbol{y}) = \|\hat{\boldsymbol{y}} - \boldsymbol{y}\|^2$ (or $\ell(\hat{y}, y) = (\hat{y} - y)^2$ for scalar).

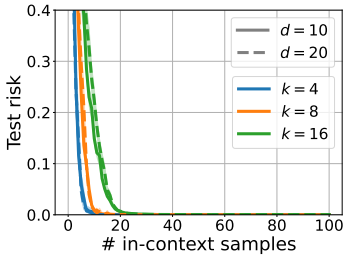


(a) CoT-I/O: 1st layer prediction

(b) Train ICL with $(\boldsymbol{x}, \boldsymbol{s})$ pairs



(c) CoT-I/O: 2nd layer prediction



(d) Train ICL with $(\boldsymbol{s}, y)$ pairs

Figure 9: We compare the performance of filtered CoT and ICL. In Fig. 9(a)&9(c), we decouple the composed risk of predicting 2-layer MLPs into risks of individual layers (following Section 3.1), which shows the filtered CoT results. In Fig. 9(b)&9(d), we train two additional models using ICL method taking $(\boldsymbol{x}, \boldsymbol{s})$ and $(\boldsymbol{s}, y)$ as inputs.

C Additional Experimental Results

C.1 Filtering evidence in 2-layer MLPs

Section 4.3 has demonstrated the occurrence of filtering in the linear deep MLPs setting (black dotted curves in Fig. 7(b)). In this section, we present further empirical evidence based on the 2-layer MLPs setting discussed in Section 4.2.

Follow the same setting as Figure 2(a) and choose $d = 10$ and $k = 8$. Assume we have a model pretrained using CoT-I/O method. As described in Section 4.2, during training, the prompt consists of in-context samples in the form of $(\boldsymbol{x}, \boldsymbol{s}, y)$ where $\boldsymbol{s} = (\boldsymbol{W}\boldsymbol{x})_+$ and $y = \boldsymbol{v}^\top \boldsymbol{s}$. To investigate filtering, we make three different predictions to evaluate the intermediate output, whose test prompts have in-context examples with the following forms:

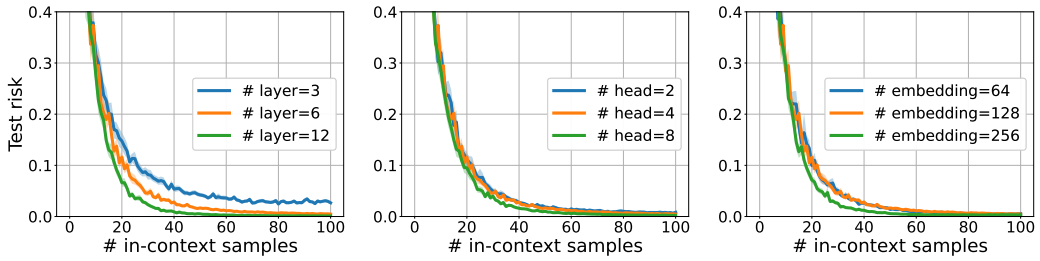
$$\text{CoT: } (\boldsymbol{x}, \boldsymbol{s}, y), \quad \text{CoT-zero: } (\boldsymbol{x}, \boldsymbol{s}, 0), \quad \text{CoT-random: } (\boldsymbol{x}, \boldsymbol{s}, z),$$

where $z \sim \mathcal{N}(0, d)$. The results are displayed in Figure 8(a) where blue, orange and green curves represent first layer prediction results using CoT, CoT-zero and CoT-random prompts, respectively. From this figure, we observe that the three curves are well aligned, indicating that when making a prediction for input $\boldsymbol{x}$, TF will attend only to $(\boldsymbol{x}, \boldsymbol{s})$ and ignore y. Therefore filling the positions of y with any random values (or zero) will not change the performance of first layer prediction.

C.2 Comparison of filtered CoT with ICL

Until now, many experimental results have shown that CoT-I/O provides benefits in terms of sample complexity and model expressivity compared to ICL. As an interpretation, we state that CoT can be decoupled into two phases: *Filtering* and *ICL*, and theoretical results have been provided to prove this statement. As for the empirical evidence, Sections 4.3 and C.1 precisely show that filtering does occur in practice. In this section, we provide additional experiments to demonstrate that, after filtering, CoT performs similarly to ICL.

For convenience and easier comparison, we repeat the same results as Fig. 2(a) in Fig. 8(b), where $d \in \{10, 20\}$, $k \in \{4, 8, 16\}$, and train with a small GPT-2. We again recap the data setting for



(a) Varying GPT-2 layers (b) Varying GPT-2 heads (c) Varying embedding dimensions

Figure 10: To further investigate how model architectures impact the prediction performance, we fix the number of heads and embedding dimension in Fig. 10(a) and change the layer number in $\{3, 6, 12\}$. Similarly for Fig. 10(b)&10(c) but instead, change number of heads (in $\{2, 4, 8\}$) and embedding dimensions (in $\{64, 128, 256\}$).

the 2-layer MLP, where the in-context examples of CoT prompt are in the form of $(\boldsymbol{x}, \boldsymbol{s}, y)$. Given that filtering happens, we make first and second layer predictions following Section 3.1 and results are presented in Fig. 9(a) and Fig. 9(c), respectively. These results show the performances of the filtered CoT prompts. Next, we need to compare the performance with separate ICL training. To achieve this goal, we train a small GPT-2 model using ICL method with prompt containing $(\boldsymbol{x}, \boldsymbol{s})$ pairs (first layer). The test results are shown in Fig. 9(b). Additionally, in Fig. 9(d), we train another small GPT-2 model using ICL but with prompts containing $(\boldsymbol{s}, y)$ pairs (second layer). By comparing Fig. 9(a) and 9(b), as well as Fig. 9(c) and 9(d), we observe that after filtering, CoT-I/O achieves similar performance as individually training a single-step problem through the ICL phase.

C.3 CoT across different sizes of GPT-2

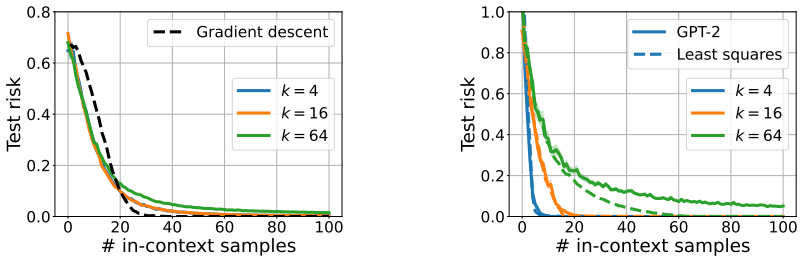
In Figure 4, we have demonstrated that larger models help in improving performance due to their ability of solving more complex function sets. However, since tiny, small and standard GPT-2 models scale the layer number, head number and embedding dimension simultaneously, it is difficult to determine which component has the greatest impact on performance. Therefore in this section, we investigate how different components of transformer model affect the resulting performance by run CoT-I/O on various GPT-2 architectures.

We maintain the same setting as in Section 4.2, fix $d = 10$ and $k = 8$, and consider a base GPT-2 model (small GPT-2) with 6 attention layers, 4 heads in each layer and 128-dimensional embeddings. In Fig. 10(a), we fix the number of heads at 4 and the embedding dimension at 128, while varying the number of layers in $\{3, 6, 12\}$. Similarly, we explore different models with different numbers of heads and embedding dimensions, and the results are respectively presented in Fig. 10(b) and 10(c). Comparing them, we can observe the following: 1) once the problem is sufficiently solved, increasing the model size does not significantly improve the prediction performance (see Fig. 10(b)&10(c)); 2) the number of layers influences model expressivity, particularly for small GPT-2 architecture (see Fig. 10(a)).

C.4 Compare transformer prediction with linear regression

We also provide experimental findings to verify Assumption 1 in this section. Previous work [Giannou et al., 2023, Akyürek et al., 2022] has theoretically proven that TF can perform similar to gradient descent, and empirical evidence from [Dai et al., 2022, Garg et al., 2022, Li et al., 2023] suggests that TF can even be competitive with Bayes optimizer in certain scenario. To this end, we first repeat the same first/layer predictions from Figure 3 in Figure 11, where $d = 10$ and blue, orange and green solid curves represent the performances of $k = 4, 16, 64$ using pretrained small GPT-2 models. We also display the evaluations of gradient descent/least square solutions in dashed curves. Specifically, in Fig. 11(a) , we solve problem

$$\hat{\boldsymbol{w}}_n = \arg\min_{\boldsymbol{w}} \frac{1}{n} \sum_{i=1}^{n} \|(\boldsymbol{w}^\top \boldsymbol{x}_i)_+ - y_i\|^2 \text{ where } \boldsymbol{x}_i \sim \mathcal{N}(0, \boldsymbol{I}_d),\ y_i = (\boldsymbol{w}^{\star\top} \boldsymbol{x}_i)_+$$



(a) 1st layer: compare with GD solving ReLU (b) 2nd layer: compare with least squares

Figure 11: Fig. 11(a): compare transformer results (solid) with gradient descent optimizer (dashed) when solving first layer of 2-layer MLPs; Fig. 11(b): compare transformer result (solid) with least squares optimizer (dashed) when solving the second layer of 2-layer MLPs.

for some $\boldsymbol{w}^\star \sim \mathcal{N}(0, 2\boldsymbol{I}_d)$ and n is the training sample size. Then the normalized test risks are computed by $\mathcal{L}(n) = \mathbb{E}_{\boldsymbol{w}^\star, \boldsymbol{x}}[\|(\hat{\boldsymbol{w}}_n^\top \boldsymbol{x})_+ - y\|^2]/d$, and we show point-to-point results for $n \in [N]$ in black dashed curve in Fig. 11(a)[5]. As for the second layer, we solve least squares problems as follows

$$\hat{\boldsymbol{v}}_n = \boldsymbol{S}^\dagger \boldsymbol{y} \text{ where } \boldsymbol{S} \in \mathbb{R}^{n \times k},\ \boldsymbol{S}[i] = (\boldsymbol{W}^\star \boldsymbol{x}_i)_+,\ \boldsymbol{y}[i] = \boldsymbol{v}^{\star\top} \boldsymbol{S}[i],\ \boldsymbol{x}_i \in \mathcal{N}(0, \boldsymbol{I}_d)$$

for some $\boldsymbol{W}^\star \in \mathbb{R}^{k \times d} \sim \mathcal{N}(0, 2/k)$ and $\boldsymbol{v}^\star \sim \mathcal{N}(0, \boldsymbol{I}_k)$. Here, † represents the pseudo-inverse operator. Then we calculate the normalized test risk of least square solution (given n training samples) as $\mathcal{L}(n) = \mathbb{E}_{\boldsymbol{W}^\star, \boldsymbol{v}^\star, \boldsymbol{x}}[\|\hat{\boldsymbol{v}}_n^\top \boldsymbol{s} - y\|^2]/d$ where $\boldsymbol{s} = (\boldsymbol{W}^\star \boldsymbol{x})_+$ and $y = \boldsymbol{v}^{\star\top} \boldsymbol{s}$. The results are presented in Fig. 11(b) where blue, orange and green dashed curves correspond to solving the problem using different values of $k \in \{4, 16, 64\}$. In this figure, the curves for $k = 4, 16$ are aligned with GPT-2 risk curves, which indicates that TF can efficiently solve linear regression as a least squares optimizer. However, the curve for $k = 64$ does not align, which can be attributed to the increased complexity of the function set with higher dimensionality ($k = 64$). Learning such complex functions requires a larger TF model.

[5]To mitigate the bias introduced by ReLU activation, we subtract the mean value during prediction, *i.e.*, $\mathcal{L}(n) = \mathbb{E}_{\boldsymbol{w}^\star, \boldsymbol{x}}[\|(\hat{\boldsymbol{w}}_n^\top \boldsymbol{x})_+ - y - (\mathbb{E}_{\boldsymbol{x}}[(\hat{\boldsymbol{w}}_n^\top \boldsymbol{x})_+ - y])\|^2]/d$.