Internal Representations, Computation, and Efficiency in Transformers

by

Liu Yang

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2026

Date of final oral examination: 02/18/2026

The dissertation is approved by the following members of the Final Oral Committee:

Robert Nowak, Professor, Electrical and Computer Engineering
Dimitris Papailiopoulos, Associate Professor, Electrical and Computer Engineering
Kangwook Lee, Associate Professor, Electrical and Computer Engineering
Frederic Sala, Assistant Professor, Computer Sciences
Jiawei Zhang, Assistant Professor, Computer Sciences
Ramya Vinayak, Assistant Professor, Electrical and Computer Engineering

Acknowledgements

Completing this dissertation has been a long and rewarding journey, and I am deeply grateful to the people who guided and supported me along the way.

First and foremost, I would like to express my sincere gratitude to my advisors. I have been incredibly fortunate to have not just one, but three outstanding advisors throughout my PhD — Prof. Robert Nowak, Prof. Dimitris Papailiopoulos, and Prof. Kangwook Lee.

I feel very grateful to have Prof. Robert Nowak as my advisor, for his kindness, wisdom, and inspiring commitment to fundamental and principled research. He asks insightful questions, and is always patient and encouraging when I am stuck on a problem or struggling to articulate my ideas. What I admire most about Rob is his remarkable openness: he trusted my judgment in how I structured my time and research, giving me the freedom to explore both academic and external opportunities while providing steady guidance throughout. Beyond research, Rob has an extraordinary ability to sense when a student needs support and to simply be there. During the hardest stretches of my PhD — when I felt uncertain about which direction to explore and the frustration was mounting — Rob would always make

time to talk, and our conversations never failed to leave me re-energized. His generosity of spirit and the way he treats everyone around him set a standard I aspire to uphold in my own career.

I am also deeply grateful to Prof. Dimitris Papailiopoulos for his mentorship, boundless enthusiasm, and unwavering support throughout my PhD. Dimitris has an incredible passion for research and a sharp instinct for identifying the problems that truly matter and tackling them with simple yet intuitive solutions. Following Dimitris into the world of LLMs and in-context learning was one of the most important decisions I made during my PhD. Dimitris is also an exceptionally engaging presenter, and from him I learned how to communicate my research in a way that is precise, compelling, and clear — a skill that has been invaluable to my growth as a researcher. I am always deeply impressed by his infectious passion for research and his unique perspectives, and I hope that I can carry even a fraction of that passion and originality into my own career. That is something I will always appreciate and continue striving to learn in the years ahead.

I am sincerely thankful to Prof. Kangwook Lee, who taught me the principles of what it means to do research well. Kangwook works incredibly hard and holds a very high standard, and being around that naturally raised my own. One of the most important lessons I learned during my PhD came from Kangwook: after a meeting that had not gone well, he told me to think deeply about what it means to take proactive ownership of my research. That single piece of advice reshaped how I approach my work, and I carry it with me to this day. Many moments like this, often in small and subtle ways, gradually shaped the way I think about research.

Working with Kangwook instilled in me a mindset of rigor, discipline, and hard work — values that I believe will serve me long after this PhD. I am deeply thankful for everything Kangwook has taught me.

I am incredibly grateful to all three of my advisors. Their wisdom, their energy, and their rigor have shaped how I think and work as a researcher — and I know those lessons will stay with me going forward.

I would also like to thank the members of my qualifying, preliminary, and defense committees – Prof. Frederic Sala, Prof. Stephen Wright, Prof. Josiah Hanna, Prof. Jiawei Zhang, and Prof. Ramya Vinayak – for their time, valuable feedback, and thoughtful questions that helped me shape my research.

I also want to thank my intern mentors, whose guidance complemented my academic training in important ways. I thank Dr. Minhui Huang for offering me the opportunity to work on the Meta Ads team, where I experienced large-scale data and model training for the first time. I am grateful to Dr. Mengyue Hang, Dr. Jiayi Xu, and Dr. Yun He, who collaborated with me closely and provided valuable feedback throughout. I thank Dr. Xiaoli Gao and Dr. Hamid Eghbalzadeh for my second internship at Meta, where they offered me the opportunity to work on cutting-edge recommendation research and provided consistent mentorship throughout. During this internship, I also had the pleasure of collaborating closely with Dr. Kaveh Hassani and my intern peer Dr. Fabian Paischer. I enjoyed working with them and learned a great deal from all. These industry experiences taught me how to conduct research in an organized and efficient way, which eventually led me to my third internship at Google with Dr. Nikunj Saunshi and Dr. Sashank

Reddi, working closely alongside Dr. Elan Rosenfeld, Dr. Keivan Mohtashami, Dr. Nishanth Dikkala, and Dr. Sanjiv Kumar. Throughout my time at Google, they provided me with resources and logistical support, and most importantly, offered generous time to discuss research problems and provide feedback. Some of my fondest memories from the internship are the intellectual discussions in the meetings and reading groups. I am especially grateful to Nikunj for the almost daily one-on-one meetings and the spontaneous hallway or whiteboard discussions — these conversations helped me grow rapidly in both research and engineering.

Six years at UW-Madison have been a journey of growth both in the academic and personal sense. I am grateful for meeting and collaborating with my labmates, from whom I learned so much in both research and life, and many of whom became my friends at Madison: from Dimitris' lab: Hongyi Wang, Alliot Nagle, Shashank Rajput, Matthew Grinde, Kartik Sreenivasan, Saurabh Agarwal, Nayoung Lee, Angeliki Giannou, Vasilis Papageorgiou, Zacharias Sifakis, Zheyang Xiong, Jack Cai; from Rob's lab: Rahul Parhi, Julian Katz-Samuels, Yinglun Zhu, Greg Canal, Subhojyoti Mukherjee, Jeongyeol Kwon, Jifan Zhang, Yang Guo, Danica Fliss, Joe Shenouda, Haoyue Bai, Gokcan Tatli, Julia Nakhleh, Sean Kennedy, Bill Nguyen, Sachi Sanghavi, Rishabh Sharma; and from Kangwook's lab: Tuah Dinh, Jy-yong Sohn, Changhun Jo, Michael Gira, Ruisu Zhang, Bryce Chen, Ying Fan, Ziqian Lin, Yuchen Zeng, Jungtaek Kim, Thomas Zeng, Jongwon Jeong, Ethan Ewer, Jackson Kunde, Lynnix Zou, Chungpa Lee, Dosung Lee.

Beyond my labmates, I also want to thank the other friends who supported me throughout my PhD — for the dinners, the conversations, and the countless

moments that made Madison feel like home: Jie Sheng, Yue Jing, Huixin Kang, Shenghong Dai, Wanzhu Hou, Xue Li, Ruijie Liu, Yun-Shiuan (Sean) Chuang, Dyah Adila, Jinlang Wang; and friends and mentors I looked up to during internships: Shuai Shao, Zhe Kang, Shan Zhou, Tao Jiang, Yue Li, Shikai Qiu, Jasmine Bayrooti, Shannon Sequeira, Vinod Raman, Harshay Shah, Shahriar Noroozizadeh, Matthew Toles, Chanwut Kittivorawong, Gaurav Ghosal, Ali Behrouz, Yilan Chen, Zhiwei Xu, Haike Xu, Ting-Yun Chang.

I am also grateful to my lifelong friends, whom I have known since middle school or even primary school. We may be in different cities or even countries, but the bond we share has remained unchanged: Yujia Duan, Jiang Li, Chi Wang, Chengyin Xu, Yuchen Zhang, Jiahe Niu, Siyuan Wei, Canyang Shi, Heng Cao, Xinhao Zhao, and Xumeng Cui – our group chat always warms me up and makes me laugh; Yuhao Wang, Weikang Du, Wenguanhua Chen, Xiaoyu Dong, Xuanyi Li, whom I have known since middle school and still chat with every now and then; Xilan Dong, Zhu Zhu, Xinyi Yang, and Dian Fan, among others from my undergraduate rowing team; and Bei Ouyang — I miss all the afternoons and nights we spent walking and chatting in the park.

This journey would not have been the same without my partner, Haiqian, who is also my best friend. We had lots of fun together over the past years and I really look forward to our future.

Finally, I want to thank my parents, Xiaoling and Zhijian, who have always supported me and believed in me. I am grateful for their love and support, and I am proud to be their daughter.

Contents

Abstract

Increasing computation at inference time has proven effective for scaling transformer performance; for example, by providing more demonstrations for in-context learning or generating longer reasoning traces for chain-of-thought. In both cases, the model *thinks* in the token space, incurring substantial memory and computational costs as contexts and generated sequences grow. This raises a natural question: since transformers already compute over continuous internal representations, *can we move thinking into the transformer's latent space?*

We investigate this question through the lens of *latent thinking*, in which computation is carried out within internal representations rather than through explicit token sequences. We present three complementary studies organized around the *formation*, *iteration*, and *compression* of latent representations. The first studies how task-dependent representations (*task vectors*) form during in-context learning. We characterize the conditions under which task vectors emerge and introduce an auxiliary loss that encourages their formation, resulting in improved robustness and generalization. Complementing this, the second asks whether latent representations can be iteratively refined through architectural recurrence. On regression

tasks, looped transformers that reuse parameters across layers can emulate iterative algorithms and match the performance of standard transformers with fewer parameters. Finally, the third extends latent computation to reasoning tasks, compressing chain-of-thought traces into latent representations via a dual-mode training algorithm that alternates between soft token compression and hard token decoding, substantially reducing inference computation while achieving similar or better reasoning performance.

Together, these results indicate that transformers can perform task adaptation and reasoning in their latent space rather than through explicit token sequences, offering a viable path toward efficient inference and parameter usage.

Chapter 1

Introduction and Background

1.1 Latent Thinking and Internal Computation in Transformers

Transformer models have achieved strong performance on complex tasks by increasing the amount of computation performed at inference time. This trend is evident in both in-context learning, where model performance improves as more demonstrations are provided in the input context [17], and explicit reasoning approaches that generate long intermediate reasoning steps [146]. While these strategies enable effective task adaptation and reasoning, they also substantially increase inference-time computation and memory usage as the context length and generated sequences grow.

Importantly, these costs are driven not only by task difficulty but also by how inference-time computation is expressed. In both in-context learning and explicit

reasoning, computation is carried out through **discrete** token sequences, despite transformer models internally operating on high-dimensional continuous representations. While token-level generation is valuable for interpretability, continuous representations can naturally encode information more compactly. This raises a fundamental question: *how much task adaptation and reasoning can instead be carried out implicitly within internal representations?*

Motivated by this question, a growing body of research has explored alternative approaches that perform computation within latent representations of transformer models. We refer to this perspective as the *latent thinking* paradigm. This paradigm includes several ways of leveraging latent representations, including but not limited to iterating latent states through architectural recurrence [32, 122, 44, 176], compressing explicit reasoning traces into soft tokens [57, 50], and using training objectives that encourage internalizing computation into latent representations [33, 165]. Together, these approaches show that transformer models can perform rich inference-time computation while reducing reliance on long token sequences.

Guided by this perspective, this dissertation investigates latent thinking in transformer models through a series of focused studies on internal representations and internal computation. We organize this investigation around three aspects: how latent representations are **formed**, **iterated**, and **compressed**. The studies span both controlled in-context learning settings and reasoning tasks. These inference-time settings are introduced in section 1.2, and section 1.3 details the specific contributions of each study.

1.2 Inference-Time Computation Instances

In this section, we describe the inference-time problem settings considered in this dissertation. We focus on two settings: in-context learning from input–output examples, and reasoning via chain-of-thought.

1.2.1 In-Context Learning

In-context learning refers to the ability of a transformer to adapt to varying downstream tasks when provided with a short context of input-output pairs, without requiring any update to the model parameters. Formally, for a task defined by a function $f(\cdot)$, the model is presented with a sequence of k demonstrations

$$(\boldsymbol{x}_1, f(\boldsymbol{x}_1), \cdots, \boldsymbol{x}_k, f(\boldsymbol{x}_k)),$$

followed by a test input $\boldsymbol{x}_{\text{test}}$. The model infers the task implied by the demonstrations and applies it to the test input within a single forward pass:

$$\texttt{TF}_\theta(\boldsymbol{x}_1, f(\boldsymbol{x}_1), \cdots, \boldsymbol{x}_k, f(\boldsymbol{x}_k), \boldsymbol{x}_{\text{test}}) = y_{\text{test}}$$

Throughout the introduction, we denote the transformer model as $\texttt{TF}_\theta$ for clarity; in subsequent chapters, we use M (or M_θ) for conciseness. In this formulation, task-specific information should be extracted from the context and represented internally in order to support generalization to the test input.

A simple and analytically tractable instantiation of in-context learning arises in regression problems. This setting was first proposed and studied by Garg et al. [40], where a transformer is trained from scratch to solve the regression problem defined as $f(\boldsymbol{x}) = \boldsymbol{w}^\top \boldsymbol{x}$. These regression tasks capture key challenges of

in-context learning, including task inference, generalization beyond observed examples, and sensitivity to distributional structure [40, 3, 53], while avoiding complications introduced by discrete tokenization of outputs. Moreover, many regression problems admit well-understood algorithmic solutions, providing a useful reference point for analyzing model behavior [141, 2, 48, 38].

This in-context learning setting is used in the first part of the dissertation as a controlled setting for examining how task information is formed and iterated within the internal representations of transformer models.

1.2.2 Chain-of-Thought Reasoning

A complementary inference-time setting arises in reasoning tasks, where the model is guided to solve a problem by generating a sequence of intermediate reasoning steps. Recent advances in large reasoning models (LRMs) [52] have emphasized this approach by encouraging models to externalize multi-step computation through natural language reasoning traces, commonly referred to as chain-of-thought reasoning [146].

Concretely, given a question q, the transformer generates a sequence of n intermediate reasoning steps $(c_1, c_2, \cdots, c_n)$, followed by a final answer a:

$$\texttt{TF}_\theta(q) = (c_1, c_2, \cdots, c_n, a).$$

While effective, this formulation directly ties inference-time computation to the length of the generated reasoning trace. As problem difficulty increases, the number of intermediate tokens can grow substantially, leading to increased computational cost, memory usage, and key-value cache requirements.

In the later part of the dissertation, this reasoning-based setting is used to study how latent computation can reduce reliance on long explicit reasoning traces while maintaining strong task performance.

1.3 Summary of Contributions

This dissertation studies latent thinking in transformer models through three complementary lines of work:

Latent Task Representation Formation in In-Context Learning [162]. We study how task-dependent latent representations, which we refer to as *task vectors*, form during in-context learning. Using transformers trained from scratch on synthetic regression tasks, we show that task vectors can naturally emerge under specific conditions, but are often weakly expressed or non-locally encoded across layers. This makes it difficult to reliably extract or reuse task information for controlled generalization. To promote strong and localized task representations, we introduce an auxiliary training objective that explicitly encourages task information to be encoded at a prescribed location within the model. This approach eliminates the need for post-hoc identification of task-correlated representations and leads to improved out-of-distribution robustness while maintaining in-context learning performance. In addition, the resulting task vectors can be directly reused to enable zero-shot task inference without explicit demonstrations.

Iterative Computation with Looped Transformer Models [161]. We study how introducing architectural recurrence enables transformers to emulate iterative algorithms in in-context learning problems. While transformers have been shown to solve the regression problem in-context [40], their feedforward structure does not explicitly implement an iterative computation mechanism commonly used in classical learning algorithms. To address this limitation, we propose looped transformer architectures that reuse parameters across multiple computation steps by feeding the final-layer hidden representation back into the model. We show that this recurrence induces iterative computation in the latent space and allows looped transformers to achieve performance comparable to transformers on a range of regression tasks, while using less than 10% of the parameters. These results demonstrate that architectural recurrence provides a parameter-efficient way to support iterative latent computation in transformer models.

Latent Reasoning with Soft Token Compression [46]. We address the inference-time cost of explicit chain-of-thought reasoning by developing a training algorithm that compresses reasoning traces into latent representations. Soft tokens, defined as hidden states fed back as input, provide a mechanism for internalizing intermediate computation, but they receive no direct supervision and can underperform explicit chain-of-thought on certain tasks. We propose Stoic-Reasoner, a dual-mode training framework in which the model alternates between a latent thinking mode that generates soft tokens and a local decoding mode that produces explicit reasoning steps as hard tokens. The cross-entropy loss on these hard tokens provides the training signal to the soft tokens, implicitly teaching them to compress the

reasoning needed for subsequent decoding. We show that this approach achieves similar or better performance compared to chain-of-thought finetuning on small-scale mathematical and logical reasoning benchmarks.

Together, these studies show how latent representations can be formed, iterated, and compressed to support efficient inference-time computation and parameter usage in transformer models. When properly structured through architectural design and training objectives, they can effectively enable task adaptation and reasoning while reducing computational and memory costs.

1.3.1 Organization of the Dissertation

The remainder of this dissertation is organized as follows:

- **chapter 2** reviews background and related work on in-context learning, task representations, and prior approaches to latent and implicit computation in transformer models.
- **chapter 3** studies how task-dependent latent representations, referred to as task vectors, emerge during in-context learning and introduces a training objective that promotes localized and reusable task representations.
- **chapter 4** investigates looped transformer architectures and training algorithms that enable iterative latent computation through architectural recurrence, emulating convergent iterative algorithm on data-fitting problems.

- **chapter 5** presents a framework for latent reasoning with soft token compression, which reduces inference-time computation by internalizing explicit reasoning traces into compact latent representations.
- Finally, **chapter 6** summarizes the contributions of this dissertation and discusses limitations and directions for future work.

Chapter 2

Related Work

The previous chapter introduced the idea that inference-time computation in transformers can be carried out within internal representations rather than through explicit token sequences, organized around three studies on the formation, iteration, and compression of latent representations. This chapter reviews prior work relevant to these studies, covering task representation in in-context learning, iterative computation through recurrence and latent tokens, as well as efficiency improvements enabled by latent representations.

2.1 In-Context Learning and Task Representation

As formalized in the previous chapter, in-context learning provides a concrete setting for studying how models adapt to new tasks at inference time without parameter updates. A central question in the literature is how task structure is internally inferred and represented.

2.1.1 Understanding the In-Context Learning Behavior

In-Context Learning in Pretrained Large Language Models. Following the introduction of GPT-3 [17], which demonstrated strong few-shot performance at inference time, a substantial body of work has sought to understand the mechanisms underlying in-context learning. Several studies have explored the significance of labeling in in-context demonstrations [100, 73, 75, 92], while others have examined the phenomenon through circuit mechanisms [35, 143, 129, 56, 144]. Research by Dai et al. [28], Geva et al. [45], Merullo et al. [98] links in-context learning behavior to implicit internal weight and activation updates within the model. To better understand "learning" in-context, several works [107, 128] have studied its two distinct phases—task recognition and task learning—using controlled experiments designed to disentangle these phases. Park et al. [108] studied the phase transition along the context length and depth. Collectively, these studies suggest that in-context learning relies on internal latent computation rather than simple pattern matching, motivating the investigation of how task information is represented and processed within intermediate layers.

In-Context Learning in Controlled Small-Scale Settings. Researchers have also examined in-context learning behavior in small-scale models, where models are trained from scratch to perform in-context learning tasks. Garg et al. [40], Akyürek et al. [3], von Oswald et al. [141, 142] investigated the ability of transformers to learn regression problems, interpreting the model as performing a single step of gradient descent: von Oswald et al. [141] empirically demonstrated that when

minimizing the ℓ_2 distance between predictions and true labels, a one-layer linear self-attention (LSA) transformer learns to implement one step of gradient descent as its optimal solution for the linear function, while Ahn et al. [2], Zhang et al. [170, 171] and Mahankali et al. [95] provided theoretical explanations for it. Beyond one-layer LSA, Fu et al. [38], Giannou et al. [48] have demonstrated that multi-layer transformers can perform higher-order optimization.

Subsequent work extended these findings to different pretraining mixtures of tasks [117, 157], the ability to generalize to unseen tasks [158], explanations via the Bayesian optimal estimator [172, 11, 103], the viewpoint of generalization error [82], and the perspectives of task retrieval and task learning [86, 148].

Beyond regression, several studies have explored in-context learning abilities in other domains, such as reinforcement learning [78, 85], discrete function learning [16], factorial hidden Markov chains [153], and deterministic finite automata (DFAs) [4]. Swaminathan et al. [133] studied the ICL mechanism with the clone-structured causal graphs to understand the schema learning, retrieval and rebinding. Additionally, Chan et al. [21, 22] investigated the behavior of in-context learning and in-weight learning on the Omniglot datasets.

2.1.2 Task Vector in In-Context Learning

While prior work has extensively characterized the behavioral and mechanistic properties of in-context learning, a complementary line of research asks how task information inferred from demonstrations is internally represented by the model. In particular, recent work suggests that such information may be compressed into

a single latent vector that can be extracted, analyzed, and manipulated. From the perspective of latent thinking, this line of work supports the view that demonstrations are summarized into a compact latent state that can be reused to guide subsequent decoding.

Task Vectors as Latent Task Representations. The notion of a "task vector" was first introduced by Ilharco et al. [67] in weight space. Subsequently, Hendel et al. [59], Merullo et al. [97], Yu et al. [166], Liu et al. [89], Saglam et al. [119], Li et al. [83] demonstrated that a single vector in the model's activation space can encode learned functions in a pretrained model. Specifically, Liu et al. [89] showed that, when presented with an in-context learning demonstration, the activation at each layer points to a subspace encoding the task information. Additionally, Todd et al. [139] identified the "function vector", another form of task vector, by averaging the causal attention heads, which can also guide the pretrained language model's performance towards desired tasks. Task vectors have also been identified across different modalities [93, 62, 110] and studied from different perspectives, including cognitive science [114] and theoretical analysis [136]. Park et al. [109] observe that concepts are represented linearly as directions in some representation space.

Emergence and Formation of Task Vectors. Beyond identifying task vectors, recent work has focused on how such representations emerge and how architectural constraints influence their formation. Mittal et al. [101] investigate whether explicitly learning appropriate latent representations via bottleneck architectures improves robustness in in-context learning, while Kobayashi et al. [74] show that bot-

tlenecks can enhance compositional generalization, albeit under a notion of compositionality centered on meta-skill learning rather than the sequential compositionality studied in this dissertation. Bottleneck architectures have also been used to analyze prequential in-context learning behavior [36]. More directly, Han et al. [55] study the dynamics of task vector formation in both the sparse linear function and pretrained language models, showing that the emergence of task vectors coincides with the development of conditional decoding strategies, leading to improved in-context learning performance. These lines of work are closely related to amortization-based meta-learning frameworks [41, 151], as well as approaches that learn belief-state representations [64] or predictive state representations in reinforcement learning [88, 104], which similarly learn compact latent summaries of context to guide downstream prediction.

Applications and Variants of Task Vectors. In addition to the benefits of zero-shot task inference through task vectors, Li et al. [80] utilized this task vector to enhance test-time adaptation, while Pham et al. [112] employed it to erase malicious concepts during pretraining. Beyond activations, task information can also be encoded in a task token [12], or a pause token can be used to gain extra computation time [49]. Zhao et al. [174] propose to present each task concept with a Gaussian distribution. Following the notion of "task vector" in the weight space, several works utilize the weight space difference for multilingual chatbot adaptation [65, 72], emotion transfer [71], aesthetic assessment [168], and soft prompt initialization [14].

2.2 Depth, Recurrence, and Latent Tokens for Implicit Computation

Beyond analyzing observable behaviors such as in-context learning, a growing body of work investigates how transformers perform computation implicitly within latent representations. These approaches study mechanisms that enable additional inference-time computation without explicit token-level reasoning, including depth-dependent processing, iterative latent updates, and the use of continuous or soft tokens as carriers of intermediate computation.

2.2.1 Depth-Dependent Behavior in Language Models

Latent thinking in transformer models is closely tied to depth-dependent computation along the forward pass. Researchers have extensively studied the layer- and depth-dependent behaviors of large language models (LLMs) along the forward pass. nostalgebraist [105], Belrose et al. [15], and Wendler et al. [147] analyzed how GPT models gradually determine top tokens through the logit lens and embedding space [30, 45]. Lioubashevski et al. [87] further showed that transformers identify top-k tokens sequentially across layers. Beyond token predictions, several works investigate layer-specific encoding of information. For instance, Lv et al. [94], Yu et al. [166], Meng et al. [96], and Gurnee and Tegmark [54] explore how factual associations are stored in a layer-dependent manner. Additionally, Liu et al. [91] studied contextual sparsity changes across layers, while Sia et al. [128] observed that "task recognition" typically occurs during the middle stages of the for-

ward pass. More broadly, phase transitions in transformer behavior across layers have been discussed by Lad et al. [76].

Leveraging these insights into layer-dependent behaviors, several works propose strategies to enhance LLM performance. Sharma et al. [125] demonstrated that selectively applying low-rank decomposition in specific layers can improve reasoning ability in question-answering tasks. Meanwhile, Sia et al. [128] and Liu et al. [91] introduced methods to reduce inference computation by identifying and leveraging specific layer behaviors. These findings indicate that transformers allocate different types of latent computation to different layers, providing empirical grounding for the task vector analysis in chapter 3.

2.2.2 Looped Transformer for Iterative Computations

Looped models reuse parameters across the forward pass to iteratively refine latent representations before producing token outputs. This idea can be traced back to early work on recurrent and weight-shared transformers [32, 77, 69]. More recently, looped models have regained attention following the study of looped transformers as programmable computers [47], which demonstrates that transformers can emulate complex algorithms and programs through iterative latent computation. Since then, this paradigm has been explored on synthetic tasks such as task difficulty extrapolation [123, 37], inversion problem [106] and fixed-point approximation [161, 39, 42, 43], and has been further scaled to large-scale pretraining, where looping has been shown to improve reasoning performance [122, 44]. These findings are consistent with broader evidence that increasing effective depth or

computation improves performance on reasoning-intensive tasks [176, 120, 164]. At the extreme, implicit models [7] repeatedly apply the same function until convergence, effectively realizing infinite-depth computation; applications of such implicit formulations are discussed in the next paragraph.

Deep Implicit Model. Deep Implicit Models [6, 7, 8, 149, 10] employ black-box solvers to find fixed-point solutions for implicit deep models. Later, Bai et al. [9] proposed the Jacobian regularization to stabilize the training process. Nevertheless, this approach requires extensive hyperparameter tuning and still suffers from instability challenges. In addition, implicit models have been demonstrated to solve math problems with extrapolation ability [31]. While the looped transformer studied in chapter 4 does not employ a black-box solver, the fixed-point perspective motivates the training strategy developed there.

2.2.3 Latent Thinking via Tokens

Beyond looped transformers, latent thinking can also be implemented through the use of continuous or latent tokens as inputs to the model. The idea of augmenting LLMs' capabilities by providing special tokens in the input, such as pause tokens [49], dot tokens [111] and latent (dummy) tokens [132], has been shown to improve reasoning with little or no additional training or architectural changes. A related direction uses soft tokens, which can be created in multiple ways: (i) by superposition over input embeddings [154], (ii) with learned discrete codes by using a VQ-VAE [131], or (iii) by feeding back the final layer's hidden representations

as input [57]. Superposed soft tokens in particular can encode parallel reasoning traces, suggesting higher expressive capacity than fixed "hard" token sequences [50]. Zhang et al. [173], Wu et al. [150], Zhuang et al. [178] explored superposing embeddings according to the next token's probability distribution without any finetuning, *i.e.*, expectation over token embeddings, while other works involve some type of training to learn the superposition [135, 70, 20, 155].

In this dissertation, we focus primarily on approach (iii), which feeds the last layer's hidden representations back to the input. This idea was first investigated by Hao et al. [57], who aimed to compress explicit CoT steps into a fixed number of soft tokens, and has since been extended to pretraining [134] and theoretical analysis of expressiveness of soft tokens [175]. Initial attempts in training models to use soft tokens [57, 25] result in performance loss, relative to supervised finetuning baselines, motivating methods that improve training stability. For example, Shen et al. [127] add a distillation loss; Hwang et al. [66] train an encoder–decoder compressor and a latent-thinking model guided by the decoder; Zhang et al. [169] use custom masks (in the spirit of 102) to teach compression from hard-token sequences into soft tokens, and Yue et al. [167] explore latent thinking through reinforcement learning. Closely related, Shen et al. [126] introduces a hybrid approach that interleaves soft tokens with selectively preserved hard tokens in the context. They segment long reasoning traces into sentences or paragraphs, retaining mathematical expressions in text form, and compressing the remaining context into soft tokens, enabling efficient parallel processing during training.

Internalizing Thinking Process. Beyond using special tokens for latent thinking, another thread aims to internalize chain-of-thought (CoT) computation into latent representations. Deng et al. [33] distill hidden representations produced by explicit reasoning trajectories into standard forward passes, and Deng et al. [34] extend this by progressively removing CoT steps while preserving performance. Similarly, to transfer deliberate "System 2" computation into a faster "System 1" generation mode, Yu et al. [165], Liao et al. [84] distill step-by-step reasoning into direct inference, while Wang et al. [145] devise System 1.5 which combines latent computation with early exiting for faster inference. Complementary approaches reduce inference cost by adaptively shortening traces, e.g., Su et al. [130] strategically drop randomized traces for fast-mode inference, and Xia et al. [152] skip tokens deemed less informative during reasoning.

2.3 Latent Representations for Efficiency

From a latent thinking perspective, many efficiency gains in transformer models stem from shifting computation from explicit token sequences to compact latent representations that summarize and reuse intermediate reasoning. Below we present a few applications where inference efficiency is improved through latent thinking.

Soft Prompt and Efficient Adaptation of Transformers. The task vector observed in Hendel et al. [59] can be viewed as a form of context compression for in-context learning: information from the demonstration is summarized into a compact con-

tinuous representation that can be injected to steer inference. This idea is closely related to soft prompt [156, 72], where a frozen backbone model is conditioned by optimizing a small set of continuous prompt parameters rather than updating all weights. Concretely, prefix-tuning [81] prepends a learned sequence of "virtual" tokens to steer generation, while prompt tuning [79] demonstrates that these learned soft prompts become increasingly competitive with full finetuning as model scale grows; similarly, P-Tuning [90] uses continuous prompt embeddings to stabilize and improve prompt-based NLU. Beyond directly optimizing prompts, hypernetwork-based methods generate prompts conditioned on the task: HyperPrompt [58] (and related hyper-tuning variants) enable parameter sharing across tasks by producing task-specific soft prefixes.

Beyond learning soft prompts that encode instructions, researchers have also explored context compression by modifying the attention mask to ensure the summarization of context at a specific token: Mu et al. [102] distill prompts into compact "gist" tokens that can be cached and reused for efficiency. Similarly, Ren et al. [118] encourage summarization at designated positions via attention mask modification, and Phang [113] further studies gisting as an economical way to build hypernetworks that emit soft prefixes from few-shot inputs.

Long Context Compression. Beyond soft-prompt-based context compression in a single window, a complementary approach augments transformers with explicit mechanisms to carry information beyond the fixed context window. These approaches are closely related to recurrent and weight-sharing architectures, reusing parameters across segments while propagating latent state or memory to enable

long-sequence processing. For instance, Transformer-XL [29] caches hidden states from previous segments (across layers) and reuses them in attention for subsequent segments, enabling longer-range dependencies than a single window. Recurrent Memory Transformer (RMT) [18] instead maintains a small set of learned memory states that are carried across segments and iteratively updated; Bulatov et al. [19] scale this idea to million-token contexts. Building on this, Chevalier et al. [26] finetune pretrained models with an adaptive (rather than fixed) number of memory tokens, showing improved long-context performance. Complementary to compressing context through continuous vectors, other works aim to compress context by replacing explicit CoT with shorter verbal summaries learned during training [160, 159, 1].

Chapter 3

Task Vectors in In-Context Learning: Emergence, Formation, and Benefits

In-context learning is a remarkable capability of transformers, referring to their ability to adapt to specific tasks based on a short context. Previous research has found that task-specific information is locally encoded within models, though their emergence and functionality remain unclear due to opaque pretraining processes. In this chapter, we address the first aspect of the latent thinking framework—the *formation* of task-dependent latent representations—by investigating the formation of task vectors in a controlled setting, using models trained from scratch on synthetic datasets. Our findings confirm that task vectors naturally emerge under certain conditions, but the task information may be relatively weakly and/or non-locally encoded within the model. To promote strong task vectors encoded at a prescribed location within the model, we propose an auxiliary training mecha-

nism based on a *task-vector prompting loss* (*TVP-loss*). This method eliminates the need to search for task-correlated encodings within the trained model and demonstrably improves robustness and generalization.

3.1 Introduction

To understand the underlying mechanisms of in-context learning in transformers, researchers have probed pretrained models from various perspectives, such as altering the labels in demonstrations [100, 73] and investigating circuit mechanisms [35, 144, 56, 129]. Additionally, controlled, small-scale studies have been conducted by training transformers from scratch to observe their in-context learning behavior on linear functions [40, 141], discrete functions [16], hidden Markov chains [153], and DFAs [4]. Furthermore, theoretical approaches have also been applied to this problem [153, 86, 47].

Among the various efforts to probe pretrained models, one significant line of research employs the concept of a "task vector", which is a vector in the model's weight or activation space[1] that encodes task-specific information. The concept of the task vector was first introduced by Ilharco et al. [67], where it is defined as a direction in a model's weight space corresponding to a particular task. Subsequently, Hendel et al. [59] demonstrated that, given a task demonstration as context, a pretrained large language model forms a task vector in its activation space at certain layers. This task vector encodes only the task information and is independent of the specific demonstration of the task. By inserting the task vector directly into the

[1]The "activation space" refers to the space where the output of each transformer layer resides.

model, it is able to perform the task without context or demonstration (*i.e.*, zero-shot). We will refer to this as *Task-Vector Prompting* (TVP) in this chapter. Concurrently, Liu et al. [89], Todd et al. [139], Merullo et al. [97], Li et al. [83], Saglam et al. [119] have also identified a single vector that encodes the task information, albeit using different terminology. We omit the details here and refer the reader to the related work and the original papers for more information.

Motivated by prior observations of task vectors in pretrained LLMs—where training conditions are inherently difficult to control—we investigate their emergence in a controlled setting by training transformers from scratch on synthetic datasets. As shown in fig. 3.1, when trained from scratch on the linear function defined as $f(\boldsymbol{x}_i) = \boldsymbol{w}^\top \boldsymbol{x}_i$, the transformer (dashed line) demonstrates the ability to use in-context learning (ICL) to solve the task. We evaluate the trained model's performance in task-vector prompting (TVP) mode, where the task vector is extracted from the in-context learning mode and injected back in a zero-shot manner, as defined by Hendel et al. [59]. The TVP performance is much better than chance (indicated by the horizontal dashed line), but lower than ICL performance, which we attribute to the fact that the encoding of the task may not be strong and localized under vanilla training methods. To encourage the formation of a strong and localized task vector, we propose an auxiliary training loss, called the *task-vector prompting loss* (*TVP-loss*). As a result, the model is trained using the TVP-loss in addition to normal training losses. In fig. 3.1, our approach (solid line) achieves comparable ICL performance to the vanilla model in the ICL mode and its TVP performance is significantly improved and comparable to ICL performance with

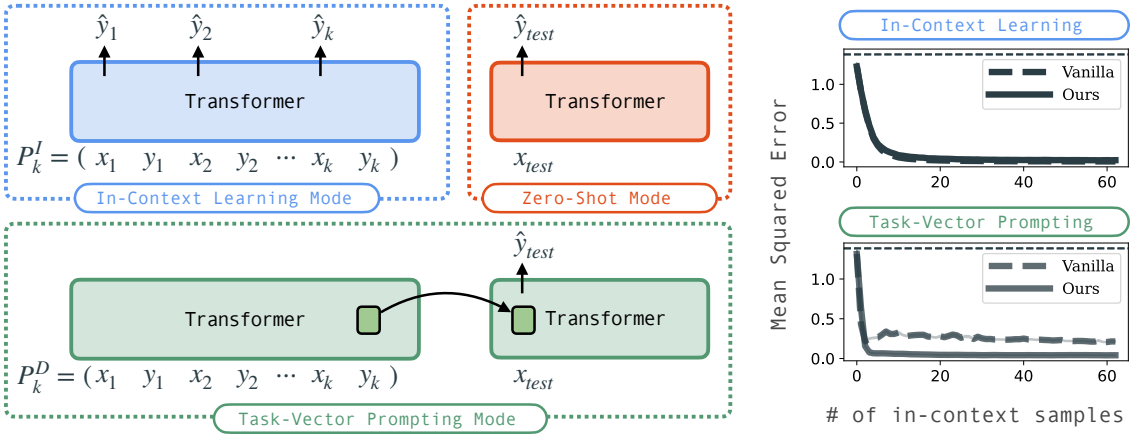


Figure 3.1: *Overview of the Transformer Operating in In-Context Learning (ICL) and Task-Vector Prompting (TVP) Modes.* A transformer can be configured to operate in ICL mode, using input-output pairs as prompts, or in TVP mode, extracting task-specific embeddings for zero-shot predictions (architecture and training details in section 3.2). On the right, the ICL and TVP performances of the vanilla-trained model and our method are shown, with the dashed horizontal line indicating random prediction performance (*i.e.*, no task information is inferred). Compared to vanilla training, our approach enhances task-specific representations in the TVP mode while preserving comparable ICL performance.

multi-shot demonstrations.

Our main contributions and findings are outlined below:

Emergence of Task Vectors During Training. We investigate the effectiveness of task vector extraction methods–originally proposed for pretrained LLMs–when applied to small-scale models trained from scratch on synthetic datasets. Our findings show that task vectors can naturally emerge during training, consistent with the observations in Hendel et al. [59] for pretrained LLMs, provided appropriate input formats and sufficient model capacity. To better understand what promotes this emergence, we examine the role of model depth and in-context length.

Strong Task Vectors with Proposed Auxiliary Loss. Although task vectors naturally emerge, they are often weak and entangled with information from input queries. This prevents task vectors from representing purely task-specific knowledge. To overcome this limitation, we propose a training algorithm that explicitly encourages the formation of task vectors independent of query information. This approach strengthens the task vector, ensuring that task-specific encoding is explicitly established within the model.

Task Vectors for In-Context Learning Robustness. Using our proposed training algorithm, we analyze the effects of TVP-loss on (a) synthetic, (b) formal language, and (c) natural language tasks, demonstrating enhanced robustness in in-context learning.

3.2 Task Vector Definition

In this section, we formally define the notion of the task vector. Let $\mathcal{F}$ denote a class of functions or "tasks", and let $\mathcal{X}$ and $\mathcal{Y}$ be the input and output spaces, respectively. If $f \in \mathcal{F}$ is the task in a specific "context", then for any input $\boldsymbol{x} \in \mathcal{X}$, the corresponding output is $\boldsymbol{y} = f(\boldsymbol{x}) \in \mathcal{Y}$. Consider $\boldsymbol{x}_{\text{test}} \in \mathcal{X}$, and the transformer model M. We can measure the model's zero-shot performance by $\ell(f(\boldsymbol{x}_{\text{test}}), M(P_{\text{query}}))$, where $P_{\text{query}} = [\boldsymbol{x}_{\text{test}}]$ denotes the query input. The in-context learning performance with k-shot examples is measured by $\ell(f(\boldsymbol{x}_{\text{test}}), M(P_k))$, where P_k is the k-shot in-context prompt $[\boldsymbol{x}_1, f(\boldsymbol{x}_1), \cdots, \boldsymbol{x}_k, f(\boldsymbol{x}_k), \boldsymbol{x}_{\text{test}}]$.

Consider a *demonstrated prompt* P_k^D, where the superscript D highlights its role

in demonstrating task information. This prompt contains k in-context samples: $P_k^D = [\boldsymbol{x}_1, f(\boldsymbol{x}_1), \cdots, \boldsymbol{x}_k, f(\boldsymbol{x}_k)]$ A task vector is an internal embedding $\boldsymbol{\tau}$ extracted when the model is presented with P_k^D, which encodes the task at hand (f). Inserting the task vector into the model during zero-shot prompting is denoted by $M(P_{\text{query}}; \boldsymbol{\tau})$. Note that the task vector $\boldsymbol{\tau}$ is extracted from the internal embeddings when inputting P_k^D, not P_k, therefore the task vector $\boldsymbol{\tau}$ does not have explicit knowledge of $\boldsymbol{x}_{\text{test}}$ when encoding the task.

A task vector extractor g tries to locate this single embedding $\boldsymbol{\tau}$ in the model given the demonstrated prompt P_k^D, *i.e.*, $\boldsymbol{\tau} = g(M(P_k^D))$. Then the performance of this extractor can be measured by $\ell(f(\boldsymbol{x}_{\text{test}}), M(P_{\text{query}}; \boldsymbol{\tau}))$. A task vector is considered successfully formed in the model M for the task f if the performance of the extracted task vector, *i.e.,* task-vector prompting performance, closely aligns with the in-context learning performance and outperforms zero-shot performance.

Hendel et al. [59] confirmed that, in pretrained large language models, $g(M(P_k^D))$ corresponds to the output embedding at approximately the middle layers of the model during the forward pass. Similarly, Sia et al. [128] found that the task in context is recognized by the model during the middle stage of the forward pass.

3.3 Emergence of Task Vectors in Trained-from-Scratch Models

3.3.1 Experimental Setup

We study task vector localization using a GPT-2 decoder trained on randomly generated prompts for the following tasks (details in section 3.7.1):

1. *Linear Function.* $f(\boldsymbol{x}_i) = \boldsymbol{w}^\top \boldsymbol{x}_i$ with $\boldsymbol{x}_i, \boldsymbol{w} \in \mathbb{R}^d$. We set $d = 6$ in our experiments.

2. *Sinusoidal Function.* $f(\boldsymbol{x}_i) = \sin(0.5 \cdot \boldsymbol{w}^\top \boldsymbol{x}_i + b)$, where $b \sim \mathcal{N}(0, 1)$.

During training, prompts are generated on the fly. Specifically, for each prompt, a function $f \in \mathcal{F}$ is randomly sampled according to the function distribution described for each task above. Subsequently, input tokens $\{\boldsymbol{x}_i\}_{i=1}^k$ are independently sampled from the corresponding input distribution. The function f is then evaluated on these inputs to produce the target outputs, forming the in-context learning prompt $P_k = [\boldsymbol{x}_1, f(\boldsymbol{x}_1), \ldots, \boldsymbol{x}_k, f(\boldsymbol{x}_k), \boldsymbol{x}_{\text{test}}]$. Let the distribution of such prompts be denoted as $\mathcal{P}$. The transformer M parameterized by θ is then trained to minimize the following expected loss:

$$\min_\theta \mathbb{E}_{P_k \sim \mathcal{P}} \sum_{i=1}^{k} \ell(f(\boldsymbol{x}_{i+1}), M_\theta(P_i)),$$

where $P_i = [\boldsymbol{x}_1, f(\boldsymbol{x}_1), \ldots, \boldsymbol{x}_i, f(\boldsymbol{x}_i), \boldsymbol{x}_{i+1}]$ represents the prompt prefix containing i in-context examples. The loss function $\ell(\cdot, \cdot)$ is defined as mean squared error. In our experiments, we use a GPT-2 model with an embedding size of 64, 4 attention

heads, and 3 layers, trained with a maximum context length of 63 (*i.e.*, $k = 63$) and without positional embeddings (NoPE). We use the Adam optimizer with a learning rate of 0.0001 and a batch size of 256, training for 300k iterations. With dynamically generated prompts, this corresponds to 76.8 million distinct prompts. All experiments are run on an NVIDIA GeForce RTX 3090.

We investigate various input formats and task vector extraction methods (details in section 3.8.1) to identify conditions under which task encoding emerges in trained-from-scratch transformers. Based on this investigation, we focus on the following setup, where task vectors emerge most distinctly:

- *input format*: Prompts with k in-context examples are generated as

$$P_k = [\boldsymbol{z}, \boldsymbol{x}_1, f(\boldsymbol{x}_1), \ldots, \boldsymbol{x}_k, f(\boldsymbol{x}_k), \boldsymbol{x}_{\text{test}}],$$

  where $\boldsymbol{z}$ is a special token placed at the beginning of the prompt to serve as a placeholder for injecting task encoding during zero-shot task-vector prompting. During training, the embedding of $\boldsymbol{z}$ is treated as a learnable parameter that is shared across all prompts (details in section 3.8.1).

- *task vector extractor method*: inspired by Hendel et al. [59], we locate the task vectors *at the activations of the token* $\boldsymbol{z}$ *and* $\{y_i\}$ in the input format mentioned above. Specifically, during task-vector prompting, the embedding is copied into the corresponding position in a zero-shot forward pass, and the zero-shot loss is measured. A lower zero-shot loss indicates that the embedding effectively encodes task-specific information (detailed in section 3.8.1).

As noted in Hendel et al. [59], the extraction process involves identifying the layer that optimally encodes the task vector. The optimal layer index $l^\star$ is determined by evaluating task vectors extracted from each layer and selecting the one that minimizes the average loss ℓ across all N demonstrated prompts. This procedure ensures that the selected layer provides the most effective task vector for predicting the test outputs. A formal description of this procedure is provided in section 3.8.2.

Remark 3.3.1 (Task Vector Location). The work of Hendel et al. [59] employs an in-context learning prompt format structured as

$$P_k = [\boldsymbol{x}_1, \boldsymbol{z}, f(\boldsymbol{x}_1), \cdots, \boldsymbol{x}_k, \boldsymbol{z}, f(\boldsymbol{x}_k), \boldsymbol{x}_{k+1}, \boldsymbol{z}],$$

where $\boldsymbol{z}$ indicates the "maps-to" token. This setup differs from the prompt format used in our experiments. In section 3.8.1, we investigate various input formats to determine their effect on task vector emergence. We find that the trained-from-scratch model exhibits task vector emergence only when the input format for training is $P_k = [\boldsymbol{z}, \boldsymbol{x}_1, f(\boldsymbol{x}_1), \cdots, \boldsymbol{x}_k, f(\boldsymbol{x}_k), \boldsymbol{x}_{k+1}]$, where no additional tokens are placed between $\boldsymbol{x}$ and $f(\boldsymbol{x})$.

3.3.2 Task Vector Emergence

Finding

Task vectors naturally emerge in small models trained from scratch.

In fig. 3.1, we demonstrate the presence of task encoding in the activations of

token y_i for the linear function, using the experimental setup described in section 3.3.1 with number of demonstrated prompts $N = 50$. In the task-vector prompting mode, the transformer achieves a mean squared error (MSE) of approximately 0.25, significantly lower than the random prediction baseline of around 1.2, indicating that the model successfully infers task information in its representation. Throughout the chapter, we use $N = 50$ by default unless stated otherwise. Additionally, we examine the impact of different N values on task-vector prompting performance, as detailed in section 3.8.3.

To understand the non-random performance observed in the task-vector prompting mode, we analyze how the model encodes task information using attention maps and PCA of activations, as shown in fig. 3.2. Attention maps quantify the relationships between input tokens: the query vector represents the current focus of interest, while the key vectors interact with the query to calculate importance scores (via their dot product), determining how relevant each input element is to the query.

For the linear function using a 3-layer transformer with a 63-shot context, the attention map of a specific head in the 2nd layer shows that the activations at the $\boldsymbol{x}_i$ positions primarily attend to the activations of the preceding y_{i-1}, where task information is stored. Furthermore, the activations at the y_i positions attend to both themselves and the preceding y_{i-1}, enabling the model to update task information incrementally. The PCA of the activations for token y_i across layers reveals that at the input to the third layer, the activations for different tasks begin to form distinct clusters, indicating that the model progressively encodes task information

as depth increases.

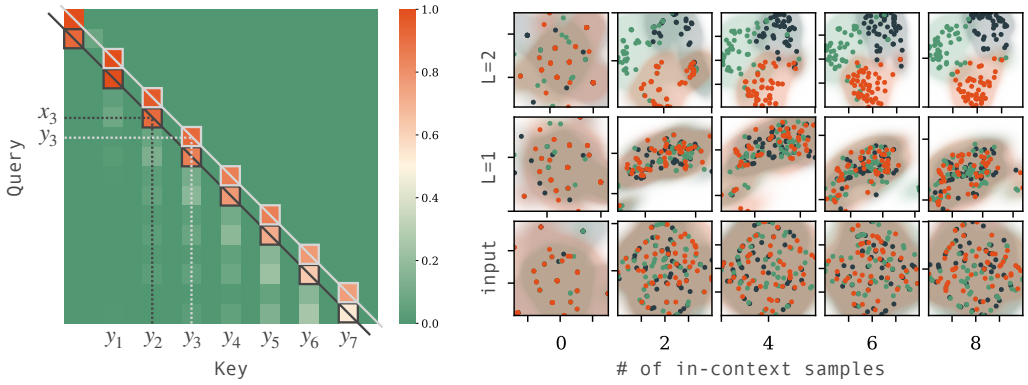


Figure 3.2: *Attention Map and PCA Visualization of Activations Across Three Different Linear Functions.* (**Left**) For the linear function described in section 3.3.1, the attention map illustrates how each query (rows) attends to all available keys (columns), with each row summing to 1. The heatmap reveals that the activations at the $\boldsymbol{x}_i$ positions predominantly attend to the activations of the preceding y_{i-1}, where task information is stored (highlighted by the black boxes). Additionally, the activations at the y_i positions attend to both themselves and the preceding y_{i-1}, enabling the online updating of task information (highlighted by the white boxes). (**Right**) PCA visualizations of token y_i activations ($i \in \{0, 2, 4, 6, 8\}$) across layers L reveal that task-specific clusters (three colors correspond to three different tasks) begin to emerge at the output of the 2nd layer, indicating that the model progressively encodes task information as depth increases.

Task Vector Layer Localization

Finding

The layer at which task vectors emerge depends on the task.

The PCA analysis above suggests that task information concentrates at the 2nd layer of the 3-layer transformer. To quantify this, we evaluate TVP performance when the task vector is extracted from each layer independently, averaging across context lengths. As shown in fig. 3.3 (bottom left), the 2nd (penultimate) layer

yields the best TVP performance, confirming it as the primary task vector location. The layer usage distribution (top left) corroborates this: layer 2 is almost exclusively selected as the best-performing layer across all context lengths, indicating that task information is stably localized at this layer regardless of how many demonstrations are provided.

This contrasts with observations in pretrained LLMs by Hendel et al. [59], where task vectors emerge in the early layers out of around 20 total layers. In our shallow models, task vectors consistently reside in the penultimate layer.

One factor to consider is the nature of the task at hand. For the linear function class, the "task" appears to be identifying the parameter $\boldsymbol{w}$, after which the "task execution" phase simply requires the model to compute $\boldsymbol{w}^\top \boldsymbol{x}$ to generate predictions. To explore this further, we examine the sinusoidal function using the same GPT-2 model configured with an embedding size of 64, 4 attention heads, and 6 layers.

As shown in fig. 3.3 (bottom right), for this 6-layer transformer model, the 3rd layer yields the best TVP performance, while other layers remain near or above the random-guess baseline. The layer usage distribution (top right) confirms this: layer 3 is selected as the best-performing layer for nearly all context lengths. Unlike the linear case, the task vector no longer resides in the penultimate layer but in an intermediate layer, suggesting that the emergence layer depends on the complexity of the task being performed.

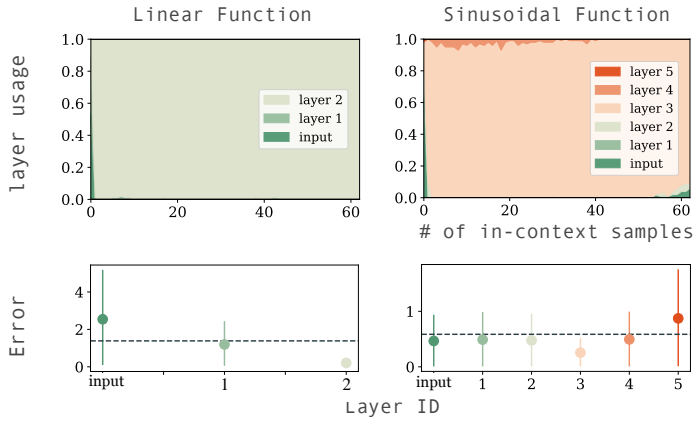


Figure 3.3: *Task-Vector Prompting Performance for Task Vector Emergence in: Linear Function* (*Left*) *and Sinusoidal Function* (*Right*). Top: layer usage distribution as a function of in-context samples, showing which layer's activation is selected as the task vector for each prompt at different context lengths. Bottom: averaged task-vector prompting (TVP) performance for each layer, measured across various context lengths. The dashed line represents random-guess performance, indicating no task information is inferred.

Effects of Model Depth and Context Length on Task Vector Emergence

For the linear function, we examine how model depth and context length affect the emergence of task vectors by varying the problem dimension and model depth. We explore $d \in \{4, 5, 6, 7, 8, 9\}$, keeping the transformer's embedding size at 64 and adjusting the model depth from $L = 3$ to $L = 8$. The transformer's performance in ICL mode (solid line) and TVP mode (line with triangular markers) is shown in fig. 3.4, with $N = 50$ for clarity.

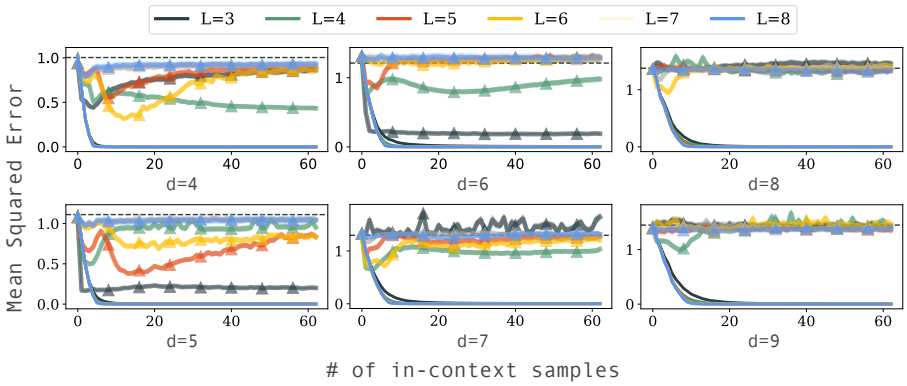


Figure 3.4: *Performance of the Transformer in In-Context Learning (ICL) Mode (Solid Line) and Task-Vector Prompting (TVP) Mode (Line with Triangular Markers) Across Problem Dimensions* $d \in \{4, 5, 6, 7, 8, 9\}$ *and Varying Model Depths* $L \in \{3, 4, 5, 6, 7, 8\}$. The dashed line indicates the random-guess baseline. The results indicate that smaller problem dimensions ($d = 4$ to $d = 7$) and shallower model depths ($L = 3$ to $L = 5$) yield stronger and more stable task encoding in the TVP mode. However, task encoding remains noisy in most cases. Task vectors emerge more clearly when the ICL loss plateaus but tend to disperse with increasing in-context length. In deeper models, task information appears to distribute across layers, reducing the distinctiveness of task vectors.

Finding

Task vectors are most distinct at moderate model depth and just before the in-context learning loss plateaus.

We observe the following characteristics of task vectors:

Along the Model Depth: Tasks with smaller problem dimensions ($d = 4$ to $d = 7$) exhibit a clear pattern of specific model depths producing strong and stable task encoding, as illustrated in the plot. However, in most cases, task encoding remains noisy. Notably, as model depth increases, the clarity of task encoding diminishes. This could be due to the model's increased capacity to approximate the least square

solution directly in a single forward pass, reducing its dependence on previously calculated task encoding.

Across the In-Context Length: Task vectors become more distinct as the in-context learning loss plateaus, but they tend to disperse afterward. For example, when $d = 5$ with $L = 4$ or $d = 7$ with $L = 4, 5, 6$, task encoding is most evident around the 5-th in-context example but then disperses into random task encodings. This suggests that the model initially learns the task in a compact and focused manner; however, over time, the task information becomes distributed more broadly across the context.

3.4 A New Training Algorithm to Encourage the Formation of Task Vector

As evidenced in the previous section, it is difficult to locate a single vector in the trained-from-scratch small-scale transformer model that cleanly encodes the task at hand. In this section, we propose a training algorithm that explicitly encourages the formation of task vectors (as defined in Hendel et al. [59]) in the in-context learning process.

Through this algorithm, which is a straightforward extension of the task vector definition, we obtain a model with the task vector explicitly formed. As demonstrated in section 3.4.2, comparing this model to the vanilla model (where task vectors are not explicitly formed) reveals that task vector formation enhances the

model's robustness for out-of-distribution in-context learning tasks. This highlights the value of task vector formation in improving generalization and interpretability in in-context learning.

3.4.1 A New Training Algorithm to Encourage Task Vector Formation

To encourage the formation of task vectors, we include the performance metric of the task vector in the training loss. Specifically, following the notation in section 3.3, let each random prompt be $P_k = [\boldsymbol{z}, \boldsymbol{x}_1, f(\boldsymbol{x}_1), \cdots, \boldsymbol{x}_k, f(\boldsymbol{x}_k), \boldsymbol{x}_{k+1}]$ to be sampled following the prompt distribution $\mathcal{P}$, and a random test prompt $P_{\text{query}} = [\boldsymbol{z}, \boldsymbol{x}_{\text{test}}]$. We train the transformer M (parameterized by θ) by optimizing the following loss:

$$\min_{\theta} \mathbb{E}_{P\sim\mathcal{P}} \sum_{i=1}^{k} \left[\underbrace{\ell(f(\boldsymbol{x}_{i+1}), M_\theta(P_i))}_{\text{ICL-loss}} + \underbrace{\ell(f(\boldsymbol{x}_{\text{test}}), M_\theta(P_{\text{query}}; h_i^l))}_{\text{TVP-loss}} \right], \tag{3.1}$$

where h_i^l represents the hidden state at the l-th layer of the transformer, extracted from the i-th in-context example. Instead of adaptively identifying the layer where the task vector resides, we simplify the process by directly designating a specific layer l as the location for forming the task vector. The hidden state h_i^l at this prescribed layer then serves as the task vector $\boldsymbol{\tau}$ defined in section 3.2. This ensures consistent representation of task-specific information and facilitates training. We illustrate the training algorithm in fig. 3.5. In section 3.4.2, we further explore the impact of this hyper-parameter choice on the model's performance.

The loss comprises two complementary components:

(1) **In-Context Learning Loss (ICL-loss)**: The term $\ell(f(\boldsymbol{x}_{i+1}), M_\theta(P_i))$ trains the model to predict the output $f(\boldsymbol{x}_{i+1})$ for the $(i+1)$-th example, based on the preceding context P_i. Summing over k examples ensures the model learns from the entire context effectively. (2) **Task-Vector Prompting Loss (TVP-loss)**: The term $\ell(f(\boldsymbol{x}_{\text{test}}), M_\theta(P_{\text{query}}; \boldsymbol{\tau}))$ with $\boldsymbol{\tau} = h_i^l$ evaluates the model's ability to use the injected task vector h_i^l, derived from the hidden state at the i-th in-context example, to predict the test output $f(\boldsymbol{x}_{\text{test}})$. This term encourages the model to encode task-specific information in the task vector $\boldsymbol{\tau}$.

As evidence of this, in fig. 3.1, although the model trained with ICL-loss already exhibits emergence of task vector, the model with TVP-loss further improves the encoded task vector, achieving a lower TVP error.

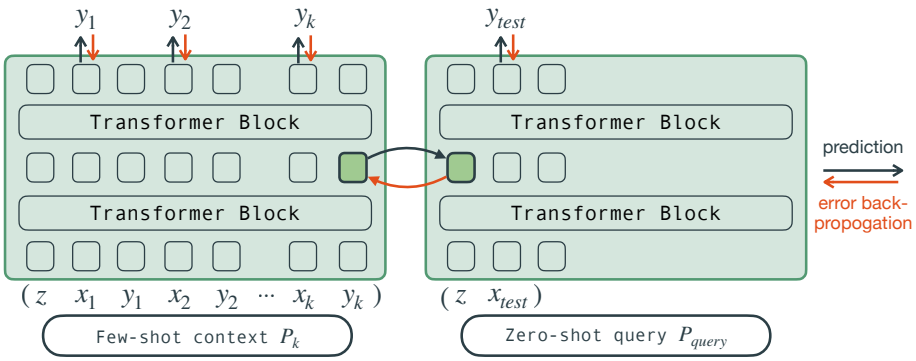


Figure 3.5: *Demonstration of Our Training Algorithm*. In vanilla Meta-ICL training, the model is updated using the ICL-loss signal from the *few-shot context*. To encourage the formation of task vectors, we also explicitly include the TVP-loss from the *zero-shot query*. This means the model is asked to predict y_{test} when only $\boldsymbol{x}_{\text{test}}$ and the injected hidden states are given. In the given illustrated example, there are in total 2 layers in the transformer model, and we set $l = 1$ to encourage the formation of the task vector at the first transformer block's output.

3.4.2 Experimental Results on Synthetic Tasks

We apply this training algorithm to the two synthetic tasks: (1) the linear function and (2) the sinusoidal function described in section 3.3.1. For all experiments, the transformer model is configured with a total of 8 layers. For consistency with the previous results, we set $N = 50$.

Finding

TVP-loss enhances task-vector prompting performance, aligning it closely with in-context learning performance.

For clarity, we evaluate the in-context learning and task-vector prompting performance at the 63rd context (*i.e.*, the final query) for models trained with and without the proposed TVP-loss. In this experiment, we designate task vectors to form at the 1st, 3rd, 5th, and 7th layers. As illustrated in fig. 3.6, for the 8-layer transformer, the task-vector prompting performance of the vanilla-trained model is nearly random. In contrast, models trained with the TVP-loss exhibit task-vector prompting performance that closely matches their in-context learning performance, as long as the task vector layer is set to an intermediate or later layer rather than the initial layers of the model.

Two observations are worth highlighting. First, the close alignment between TVP and ICL performance indicates that the task vector formed by TVP-loss captures the essential task-specific information that the model extracts from the full in-context demonstrations. In other words, a single vector at a prescribed layer suf-

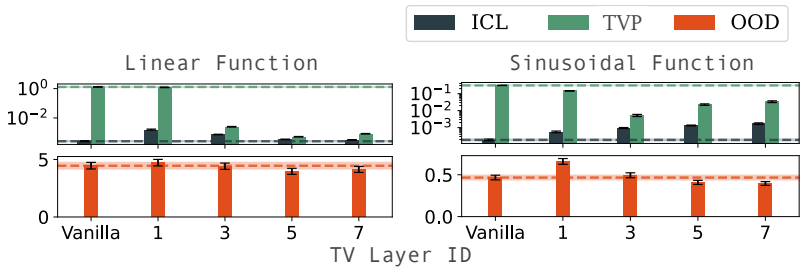


Figure 3.6: *Model Performance with Vanilla and TVP-Loss Training.* This figure compares models trained with vanilla training and TVP-loss across layers where task vectors are formed. The upper row shows in-context learning (ICL) and task-vector prompting (TVP) performance on a logarithmic scale for clarity, while the bottom row depicts out-of-distribution (OOD) prompt performance. Dashed lines represent the performance of the vanilla-trained model in each respective setting (ICL, TVP, and OOD), providing baselines for comparison. Each column corresponds to a task: linear function (left) and sinusoidal function (right). The horizontal axis indicates the layer at which task vectors are formed. **ICL performance:** Models with TVP-loss achieve similar ICL performance to vanilla-trained models. **TVP performance:** TVP-loss improves TVP performance, especially when task vectors form in intermediate or later layers. **OOD robustness:** TVP-loss enhances OOD generalization with task vectors formed at specific layers.

fices to encode the task as effectively as the entire demonstration context. Second, as shown in fig. 3.6, adding the TVP-loss to training does not interfere much with the model's in-context learning performance: across both tasks and layer configurations, models trained with TVP-loss achieve ICL performance comparable to their vanilla-trained counterparts.

Finding

TVP-loss improves in-context learning performance on out-of-distribution prompts.

Performance on OOD Prompts. Forcing the task vector to form at a specific layer can be interpreted as introducing an implicit bottleneck architecture within the

model's forward pass. In this section, we examine the model's generalization ability when the task vector is constrained to form at a particular layer. Specifically, for the two synthetic tasks, we evaluate performance on the OOD prompts described in section 3.7.1,

Note that during training, none of these OOD settings were encountered. As shown in fig. 3.6, models with task vectors formed at specific layers exhibit equal or improved performance on OOD prompts compared to vanilla-trained models. For the linear function, forming the task vector at the 5th and 7th layer yields slightly better OOD performance.

The sinusoidal function task can be considered as a compositional task, where $f_1(\boldsymbol{x}_i) = \boldsymbol{w}^\top \boldsymbol{x}_i$, and $f_2(y) = \sin(y)$. Then the OOD task modifies the $f_2(y) = \sin(y)$ to $f_2(y) = \sqrt{y}$. In this case, models with task vectors show enhanced generalization to the modified compositional task, indicating that capturing task representations at intermediate layers improves OOD generalization.

3.5 In-Context Learning Beyond Synthetic Tasks

In this section, we extend our study beyond the synthetic regression tasks and investigate the impact of the TVP-loss on synthetic formal language tasks in section 3.5.1 as well as natural language tasks in section 3.5.2.

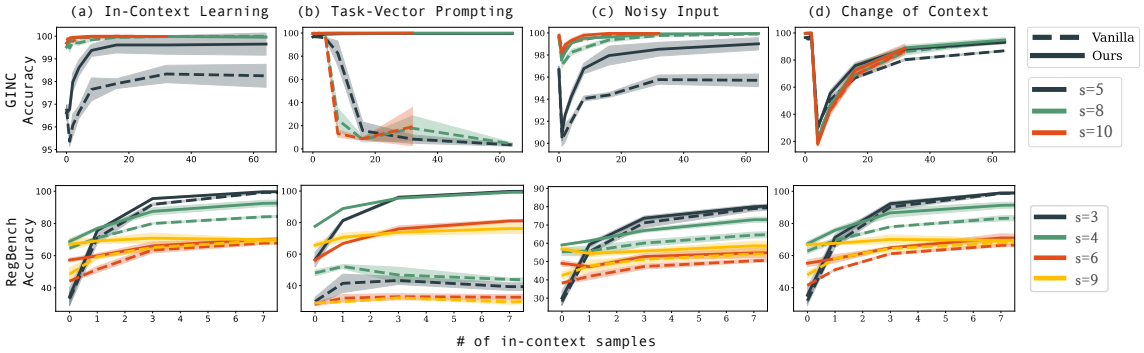


Figure 3.7: *GINC (Top) and RegBench (Bottom) Dataset In-Context Learning Performance.* Comparison of the in-context learning performance for the vanilla-trained model (dashed line) and the model trained with TVP-loss at the 5th layer for GINC and the 4th layer for RegBench (solid line). In all cases, the model with task vector formation outperforms the vanilla-trained model.

3.5.1 Synthetic Formal Language Task

We conduct experiments on two benchmarks: the Generative In-Context Learning (GINC) dataset introduced in Xie et al. [153], and the RegBench dataset from Akyürek et al. [4]. In both datasets, each context consists of s tokens per example. For instance, an in-context prompt with context length $k = 3$ and example length $s = 4$ looks like: `a b c - d / e f g - h / i j k - l`, while a corresponding zero-shot prompt is `m n o - p`.

GINC Dataset. We follow the setup in Xie et al. [153]: a uniform mixture of 5 HMMs over a vocabulary of 100 tokens. Documents of 576 tokens are generated from a sampled HMM, with 192 dummy tokens "–" inserted, yielding a total length of 768. We use a GPT-2 model with 8 layers, 8 heads, and an embedding dimension of 256. In GINC, there are no explicit input-output pairs; instead, the entire trajectory performs one task defined by a stochastic hidden Markov model.

Therefore, we encode the task vector into the inserted token "−" in the trajectory.

To construct the zero-shot document for the TVP-loss, we sample 10 extra tokens from the same HMM and randomly insert a dummy token "−". The task vector loss is computed by injecting the l-th layer's hidden states at the "−" token into the zero-shot document. For task-vector prompting evaluation, we extract hidden states at the i-th "−" token in the in-context prompt and inject them into the zero-shot prompt; throughout the experiment we set $N = 1$. We provide an illustration of our training algorithm adapted to the GINC dataset in fig. 3.8.

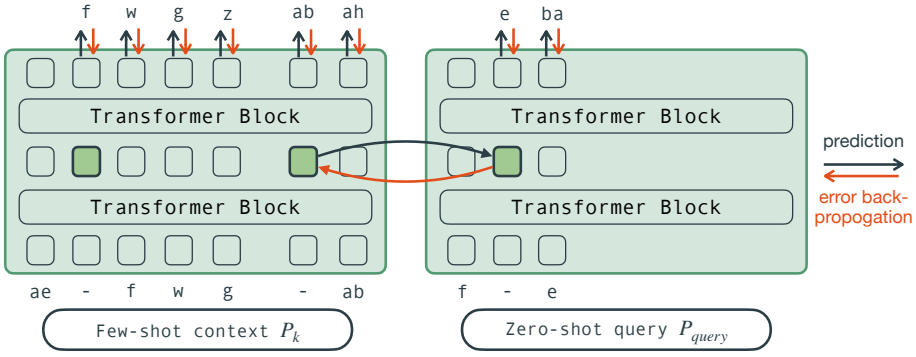


Figure 3.8: *Demonstration of Our Training Algorithm in GINC Dataset.* In contrast to the Meta-ICL format described in fig. 3.5, in GINC dataset, there are no explicit input-output pairs. Instead, the entire trajectory performs one single task defined by a hidden Markov model. Therefore, we randomly insert several "−" tokens into the trajectory in the few-shot mode, while inserting one "−" token into the trajectory in the zero-shot mode. The task vector at the l-th layer is then injected into the embedding in the zero-shot mode.

Though the model is not Meta-ICL trained—meaning the pretraining distribution differs from the prompt distribution—training with the TVP-loss still enables the formation of task vectors during in-context learning. We present the results in fig. 3.7 (top). We measure both models' performance on (a) in-context learning, (b) task-vector prompting, (c) noisy in-context learning, where the in-context

token is replaced with a random token with probability 0.1, and (d) when the underlying HMM is changed after the second context.

RegBench Dataset. We follow and revise the setup of Akyürek et al. [4]: we sample 100 deterministic finite automata (DFAs) for training, each with a vocabulary of 40 tokens, up to 20 states, and at most 10 outgoing edges per state, converted to probabilistic finite automata (PFAs) with uniform transition probabilities. Each prompt contains 32 sequences of length s separated by the special token "|", with a dummy token ">" inserted before the last token in each sequence. We use a GPT-2 model with 8 layers, 2 heads, and an embedding dimension of 128, and train with the next-token prediction loss on all tokens except the special tokens ">" and "|".

For each training sequence, we sample an additional example from the same DFA and insert a dummy token ">" before its last token; this serves as the zero-shot prompt for computing the TVP-loss. Note that, similar to GINC, the task f determined by the PFA's transition function is stochastic. During inference, we evaluate task-vector prompting with $N = 1$.

When evaluating the next predicted token in the in-context learning prompt, we follow the same setup as in Akyürek et al. [4]: as long as the next predicted token belongs to the outgoing edge set of the current DFA state, it is considered a correct prediction; otherwise, it is incorrect. Intuitively, as k increases, more transitions are presented to the transformer. This requires the model to approximate a larger n-gram distribution, leading to a degradation in performance. In fig. 3.7 (bottom), we present results on: (a) in-context learning, (b) task-vector prompting, (c) noisy in-context learning, where 4 out of 32 in-context labels are replaced

with random tokens, and (d) performance when the underlying DFA of the first 4 out of 32 samples is changed to another DFA.

Performance Analysis. Across both datasets, models trained with TVP-loss consistently outperform vanilla-trained models. Specifically, in the GINC dataset, TVP-loss improves the model's ability to learn in-context, even when trained only on next-token prediction tasks, and enables task vector formation without directly optimizing for in-context learning during pretraining. As shown in fig. 3.7, models trained with TVP-loss achieve slight improvements in ICL performance and significant gains in TVP performance. Additionally, these models exhibit increased robustness to label noise and out-of-distribution (OOD) in-context samples, consistently outperforming vanilla-trained models in OOD scenarios.

3.5.2 Natural Language Task

Motivated by the OOD improvements observed with TVP-loss on GINC and Reg-Bench, we further assess its effectiveness in natural language tasks using the Cross-Fit benchmark [163] following the MetaICL settings in Min et al. [99]. Specifically, we finetune a pretrained GPT-2 Large model (774M parameters), modifying the original MetaICL procedure by training across all context lengths rather than using a fixed context length ($k = 16$), which allows the model to generalize to varying context lengths at inference time.

Since the meta-training set contains a diverse range of tasks, the optimal layer for forming task vectors may vary across tasks. To accommodate this, we do not

fix a single task vector layer during finetuning with TVP-loss. Instead, for each training prompt, we evaluate the TVP-loss at every layer from 8 to 15 (out of 36 total layers) and select the layer that minimizes the loss. Formally, recall the TVP-loss in eq. (3.1) for layer l is defined as

$$\ell^l_{\text{TVP}} = \sum_{i=1}^{k} \ell(f(\boldsymbol{x}_{\text{test}}), M_\theta(P_{\text{query}}; h^l_i)).$$

For each training step and each prompt, instead of minimizing ℓ^l_{TVP} with predefined l, we minimize $\ell_{\text{TVP}} = \min_{l \in \{8,9,\ldots,15\}} \ell^l_{\text{TVP}}$. This allows the model to determine the most effective task vector location per prompt.

During inference, following Min et al. [99], each test query is paired with a set of candidates (e.g., classification labels or answer choices for QA), and the model selects the candidate with the highest conditional probability.

Table 3.1 presents results under three task transfer settings in the CrossFit benchmark following MetaICL settings. In each case, we train on a set of meta-training tasks and evaluate on a distinct set of target tasks to assess generalization. The three settings are:

1. High-resource → Low-resource (HR→LR): Datasets with 10,000 or more training examples—referred to as high-resource tasks—are used for meta-training, while the remaining tasks are used for evaluation.

2. Non-Class → Class: The model is trained on tasks that do not involve classification and then evaluated on classification tasks.

3. Non-Paraphrase → Paraphrase: The model is trained on tasks that do not

involve paraphrase detection and evaluated on tasks with paraphrase detection.

We report in-context learning performance (with context length $k = 16$) on the target tasks, showing the mean and standard deviation over 3 inference seeds. In addition to overall target task performance, we also separately evaluate on unseen domain target tasks–datasets whose topics (e.g., finance, poetry, climate, or medical) do not appear during meta-training–to assess cross-domain generalization. As shown in table 3.1, models meta-trained with TVP-loss show modest improvements in ICL accuracy across all three settings. We hypothesize that TVP-loss serves as a regularizer, promoting better OOD generalization by encouraging the model to form task-specific encoding.

Lastly, we evaluate the performance of task-vector prompting with $N = 10$ demonstrations. For each test query, we follow the same inference strategy as in in-context learning: pairing the query with a set of candidate answers. Instead of passing the full demonstration context, we insert the task vector extracted from the context into a specific layer determined by the N demonstrations. The task-vector prompting results are reported in the lower half of table 3.1. Although the evaluation tasks were never seen during training, TVP-loss still improves task-vector prompting accuracy in two of the three settings, while degrading in the Non-Class$\rightarrow$Class setting.

	Method	HR→LR	non-Class→Class	non-Paraphrase→Paraphrase
ICL	MetaICL	44.88 ± 0.29 / 39.14 ± 3.20	37.97 ± 1.05 / 32.71 ± 1.51	41.07 ± 2.57 / 34.05 ± 0.00
	+ TVP-loss (Ours)	$\mathbf{45.60} \pm 0.61$ / $\mathbf{42.10} \pm 3.67$	$\mathbf{39.55} \pm 1.05$ / $\mathbf{33.78} \pm 2.43$	$\mathbf{41.82} \pm 1.73$ / 34.05 ± 0.00
TVP	MetaICL	31.93 ± 0.20	$\mathbf{27.74} \pm 0.34$	30.97 ± 2.00
	+ TVP-loss (Ours)	$\mathbf{34.41} \pm 0.25$	22.71 ± 1.56	$\mathbf{33.45} \pm 0.82$

Table 3.1: *ICL and TVP Accuracy of Finetuned GPT-2 Large on the CrossFit Benchmark.* We evaluate performance under three task transfer settings: HR→LR (High-Resource to Low-Resource), non-Class→Class, and non-Paraphrase→Paraphrase. Each cell reports the mean ICL accuracy ($\pm$ std) over three inference seeds using $k = 16$ in-context examples, shown as: overall performance / performance on target tasks from unseen domains.

3.5.3 Effects of Various Hyperparameter Choices

To further understand the robustness and sensitivity of our approach, we analyze the impact of key hyperparameters on the HR→LR task:

(A) the range of candidate layers used for selecting the task vector location, and

(B) the weight w_{TVP} of the TVP-loss term, generalizing eq. (3.1) to $\ell_{\text{ICL}} + w_{\text{TVP}} \cdot \ell_{\text{TVP}}$.

Table 3.2 reports ICL and TVP performance under different configurations.

Layer Range: Expanding the candidate range from the default (8–15) to a broader span (8–18) generally improves ICL performance but degrades TVP accuracy, potentially due to noisier task vector selection when the range is too wide. Conversely, narrowing or shifting the range (6–13 or 10–17) slightly degrades both ICL and TVP performance, while the 8–12 range degrades ICL but marginally improves TVP accuracy. These results suggest that the 8–15 range strikes a balance between flexibility and task vector quality.

Loss Weight w_{TVP}: Varying the weight of the TVP-loss in the joint objective reveals a trade-off: smaller weights (e.g., $w_{\text{TVP}} = 0.3$) slightly improve ICL performance but reduce TVP accuracy, whereas larger weights (e.g., $w_{\text{TVP}} = 3$) hurt both

metrics—likely due to over-regularization. We find that $w_{\text{TVP}} = 1$ yields the best overall balance between in-context learning and task vector generalization.

Range	8-15	8-12	8-18	6-13	10-17	8-15	8-15
w_{TVP}	1	1	1	1	1	0.3	3
ICL	45.60 ± 0.61	45.33 ± 0.30	46.00 ± 0.09	45.45 ± 0.52	45.60 ± 0.47	45.93 ± 0.45	44.53 ± 0.40
TVP	34.41 ± 0.25	34.62 ± 0.33	32.91 ± 0.55	33.99 ± 0.22	33.28 ± 0.68	33.83 ± 0.10	33.65 ± 0.11

Table 3.2: *ICL and TVP Accuracy of HR→LR Task under Various Hyperparameter Setup.*

3.6 Discussion and Conclusions

Factors Affecting Task Vector Emergence. The emergence of the task vector in pretrained large language models may be attributed to multiple factors, such as the model's capacity and the diverse tasks encountered during pretraining. While Hendel et al. [59] and related studies identify and analyze the existence of task vectors, they do not study the factors that may affect their emergence and performance. This gap in understanding motivates our investigation: we reproduce this phenomenon in the small-scale setting when trained from scratch, gaining insight into the factors influencing task vector emergence, such as model depth and context length. For instance, for the linear function, a model with moderate depth encourages the emergence of task vectors, which is most evident before the in-context learning loss plateaus.

Training with TVP-Loss. While task vectors can emerge through normal training processes, adding the TVP-loss to the training process encourages the formation of task vectors at prescribed locations and leads to improved accuracy and

robustness. Across various benchmark datasets, we have demonstrated two key properties of TVP-loss training: (1) models trained with TVP-loss form clean task vectors whose prompting performance closely matches in-context learning performance, indicating that a single vector at a prescribed layer captures the essential task-specific information from the demonstrations; and (2) adding the TVP-loss does not interfere with the model's in-context learning ability, as ICL performance remains comparable to vanilla-trained models. Additionally, depending on the task, task vectors formed at intermediate or later layers can enhance the model's robustness to out-of-distribution (OOD) prompts.

Potential Applications of Task Vectors. Task vectors can serve as a soft prompt, effectively compressing the entire context into a single vector representation. In practical scenarios, where the number and nature of tasks are unknown during pretraining and only demonstrations are provided to the model, task vectors enable both context summarization and task identification. Specifically, a model with task vectors formed gains the ability to identify the underlying task from demonstrations, cluster tasks using the extracted task vectors, and subsequently perform zero-shot inference with these extracted vectors.

Another practical benefit is during inference, attention computations over earlier tokens can be masked out after the prescribed l-th layer, restricting attention to only the task vector and the query. This approach mirrors the findings of Sia et al. [128], who demonstrated a 45% reduction in computation for pretrained LLMs. Looking ahead, incorporating the TVP-loss as an auxiliary objective during training offers a promising approach to enhance task-specific representation and overall

performance.

Conclusion. We investigated how task-dependent latent representations emerge during in-context learning by training transformers from scratch on synthetic tasks. Our analysis showed that task vectors can naturally form under specific conditions, such as appropriate input format and moderate model depth, but remain weak and diffuse without explicit encouragement. To address this, we introduced the TVP-loss, an auxiliary training objective that promotes the formation of strong, localized task vectors at prescribed layers. Across synthetic, formal language, and natural language benchmarks, models trained with TVP-loss achieve comparable in-context learning performance while exhibiting substantially improved task-vector prompting accuracy and enhanced robustness to out-of-distribution prompts.

3.7 Detailed Experimental Setup

3.7.1 Synthetic Experiments

Linear Function. Following the setup in [86, 40], the input vectors $\boldsymbol{x}_i \in \mathbb{R}^d$ are sampled from a normal distribution $\mathcal{N}(0, \boldsymbol{I})$. The linear function weights $\boldsymbol{w} \in \mathbb{R}^d$, which define the task, are sampled from a mixture of d Gaussians, each with means corresponding to the standard basis vectors: $\boldsymbol{\mu}_i \in \mathbb{R}^d$ with $\boldsymbol{\mu}_{ij} = 1$ if $i = j$, and 0 otherwise. All Gaussians share a covariance matrix $\Sigma = \frac{1}{4}\boldsymbol{I}$, and each component is selected with equal probability. The target outputs are computed as $f(\boldsymbol{x}_i) = \boldsymbol{w}^\top \boldsymbol{x}_i$. We set $d = 6$ in our experiments.

Sinusoidal Function. Following the same sampling strategy as the linear function, the target outputs are defined as $f(\boldsymbol{x}_i) = \sin(0.5 \cdot \boldsymbol{w}^\top \boldsymbol{x}_i + b)$, where $b \sim \mathcal{N}(0, 1)$. In this task, $\boldsymbol{w}$ and b determine the underlying function. For the two synthetic tasks, we evaluate the following out-of-distribution (OOD) tasks:

1. *Linear Function*: We introduce an OOD prompt using the quadratic function task, defined as $f(\boldsymbol{x}_i) = \boldsymbol{w}^\top (\boldsymbol{x}_i \cdot \boldsymbol{x}_i)$, where $\cdot$ denotes element-wise multiplication.

2. *Sinusoidal Function*: we consider the OOD prompt where the task is $f(\boldsymbol{x}_i) = \sqrt{\max(0.5 \cdot \boldsymbol{w}^\top \boldsymbol{x}_i + h, 0)} + b$, where $h, b \sim \mathcal{N}(0, 1)$.

3.8 Supplements on the Trained-from-Scratch Model

3.8.1 Effects of Input Formats

In this section, we examine different input formats for in-context learning prompts and their effects on the emergence of task encoding. The various prompt formats are detailed in Table 3.3. For clarity, we refer to the token positions where task vector presence is examined as the *task token locations*.

We focus on the case where the problem dimension is $d = 6$ and the model depth is $L = 3$, as it exhibits the most pronounced task encoding, as shown in fig. 3.4. In fig. 3.9, we evaluate four task vector extraction methods, with their respective task-vector prompting performance shown as line with triangular markers. The four methods are: (A) directly using the task token's output embed-

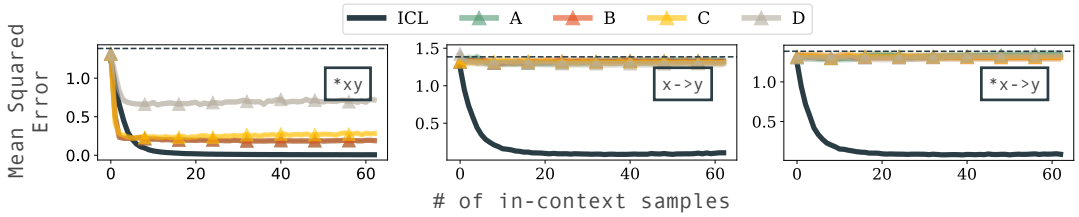


Figure 3.9: *In-Context Learning (ICL) and Task-Vector Prompting (TVP) Performance Across Input Formats and Task Vector Extraction Methods.* This figure compares task-vector prompting performance across four task vector extraction methods and three input formats: `(*xy)`, `(x->y)`, and `(*x->y)`. Method (A) is the default task vector extraction method, and `(*xy)` is the default input format used throughout the main paper. As shown, for the `(*xy)` input format, noticeable task encoding is observed, with performance exceeding random predictions, regardless of the extraction method. In contrast, for the `(x->y)` and `(*x->y)` input formats, no task encoding can be reliably extracted by any method, resulting in task-vector prompting performance that is nearly random.

ding [59], (B) using the difference between the task token's embedding and the uninformative query embedding [89], (C) taking the principal direction of this difference via PCA [89], and (D) learning a linear combination of embeddings [83]. Notably, only the model with the input format `(*xy)` demonstrates task encoding that surpasses random baselines. By default, we use the format `(*xy)` throughout the main paper unless specified otherwise.

Table 3.3: *Different Prompt Formats Examined.* We analyze the input format based on the settings described in Garg et al. [40] and Hendel et al. [59]. Here, $\boldsymbol{z}$ represents a special token, P_{query} denotes the test query format, and P_k refers to the k-shot context. We examine the activations at Λ_f, referred to as the task token location.

	P_{query}	k-shot examples P_k	Task Token Location Λ_f
`(*xy)` [40]	$[\boldsymbol{z}, \boldsymbol{x}_{\text{test}}]$	$[\boldsymbol{z}, \boldsymbol{x}_1, f(\boldsymbol{x}_1), \cdots, \boldsymbol{x}_k, f(\boldsymbol{x}_k), \boldsymbol{x}_{k+1}]$	P_k`[0::2]`: $\{f(\boldsymbol{x}_i)\}$ and $\boldsymbol{z}$
`(x->y)` Hendel et al. [59]	$[\boldsymbol{x}_{\text{test}}, \boldsymbol{z}]$	$[\boldsymbol{x}_1, \boldsymbol{z}, f(\boldsymbol{x}_1), \cdots, \boldsymbol{x}_k, \boldsymbol{z}, f(\boldsymbol{x}_k), \boldsymbol{x}_{k+1}, \boldsymbol{z}]$	P_k`[1::3]`: $\boldsymbol{z}$
`(*x->y)` Hendel et al. [59]	$[\boldsymbol{z}, \boldsymbol{x}_{\text{test}}, \boldsymbol{z}]$	$[\boldsymbol{z}, \boldsymbol{x}_1, \boldsymbol{z}, f(\boldsymbol{x}_1), \cdots, \boldsymbol{x}_k, \boldsymbol{z}, f(\boldsymbol{x}_k), \boldsymbol{x}_{k+1}, \boldsymbol{z}]$	P_k`[0::3]`: $\{f(\boldsymbol{x}_i)\}$ and $\boldsymbol{z}$

Discrepancy with Pretrained LLM

In the pretrained LLM [59], the input format uses `x->y`. There is a clear difference between the pretrained LLM and the trained-from-scratch transformer in how task information is encoded. In the pretrained LLM, task information is primarily stored in the "maps-to" token (`->`). Conversely, in the trained-from-scratch transformer, task information is stored in the y token (*i.e.*, the "label" token), but only when x and y alternate closely in the input format.

We hypothesize that this discrepancy arises because pretrained LLMs learn the semantic meaning of the "maps-to" symbol (`->`) from their extensive pretraining corpus, enabling them to use `->` as an anchor for task summarization. Additionally, this delimiter `->` is needed to separate the x and y entries. In contrast, for the trained-from-scratch transformer, the `->` token functions more like a `<pause>` token [49], providing additional computational resources rather than semantic significance. Moreover, the training procedure ensures that x and y tokens occupy fixed positions, removing the need for a delimiter to separate them.

Furthermore, Wang et al. [144] show that label words themselves can serve as anchors for aggregating task information in context. This observation aligns with our finding that task information is encoded in the y tokens. While this result is not explicitly framed within the task vector framework, it highlights the critical role of label tokens in task encoding.

3.8.2 Formal Description of Layer Localization for Task Vector

For each task f, we generate N demonstrated prompts $\{P_k^{D,j}\}_{j=1}^N$, where the same function f is consistently used across all N prompts, and the input to each prompt $P_k^{D,j}$ is uniformly sampled. Let P_{query}^j, with test input $\boldsymbol{x}_{\text{test}}^j$, represents the corresponding test queries for the j-th demonstrated prompt. For a task vector extracted from a k-shot prompt, the optimal layer index $l^\star$ is computed as:

$$l^\star = \arg\min_l \sum_{j=1}^{N} \ell \left(f(\boldsymbol{x}_{\text{test}}^j), M \left(P_{\text{query}}^j; g(M(P_k^{D,j}), l) \right) \right),$$

where $g(M(P_k^{D,j}), l)$ denotes the task vector extracted from the l-th layer when processing the demonstrated prompt $P_k^{D,j}$, and $M(P_{\text{query}}^j; \boldsymbol{\tau})$ represents the model's prediction for the query prompt P_{query}^j, where the task vector $\boldsymbol{\tau}$ extracted from the l-th layer is copied to the corresponding position in the model's representation when processing P_{query}^j. A lower loss ℓ indicates that the corresponding layer provides a more effective encoding of task-specific information.

3.8.3 Effects of Number of Demonstrated Prompts N

Throughout the paper, we use $N = 50$ by default when evaluating task-vector prompting performance. Intuitively, larger N values yield more concentrated and purified task vectors. To illustrate the effect of N, we evaluate task-vector prompting performance for $N = 1, 2, 5, 10, 50$, following the setup in section 3.3.1, and present the results in fig. 3.10. In this experiment, the transformer is trained on the linear function with a problem dimension of $d = 6$ and transformer depth $L = 3$.

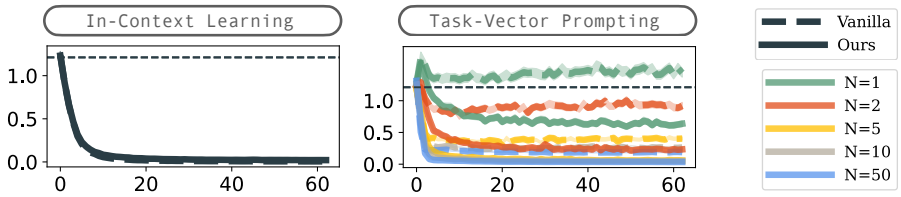


Figure 3.10: *Impact of the Number of Demonstrated Prompts (N) on Task-Vector Prompting Performance.* Increasing N improves task-vector prompting performance, as larger N values yield more concentrated task vectors. Across all N settings, model trained with TVP-loss ("Ours") consistently outperforms the vanilla model, demonstrating better task vector encoding. The dashed line represents random-guess performance.

For the model trained with TVP-loss, the task vector is explicitly formed at the 2nd layer.

As shown in the figure, when $N = 1$, the vanilla model yields random task-vector prompting performance. With increasing N, task vector performance improves steadily. Notably, the model trained with TVP-loss (denoted as "Ours" in the figure) demonstrates better task vector encoding compared to the vanilla model across all N values. Because the linear functions in regression tasks have weights that exist in a continuous space and can be very close to each other, separating tasks with only a single demonstration is challenging. As a result, it requires $N = 5$ to achieve task-vector prompting performance comparable to its in-context learning performance. For the vanilla model, however, even with $N = 50$, task-vector prompting performance remains slightly worse than in-context learning performance.

Chapter 4

Looped Transformers are Better at Learning Learning Algorithms

The previous chapter studied how task information is *formed* into latent representations during in-context learning, a setting in which transformers have demonstrated strong effectiveness in solving data-fitting problems [40]. A natural follow-up question is whether these latent representations can be progressively *refined*: in traditional machine learning, solutions can be obtained through iterative algorithms such as gradient descent, which improve an estimate over multiple steps. However, the transformer architecture processes inputs in a single forward pass and lacks an inherent iterative structure, making it difficult to emulate such iterative algorithms. This chapter investigates how latent computation can be *iterated* through architectural recurrence. We propose the utilization of a *looped* transformer architecture and its associated training methodology, with the aim of incor-

porating iterative characteristics into the transformer. Experimental results suggest that the looped transformer achieves performance comparable to the transformer in solving various data-fitting problems, while utilizing less than 10% of the parameter count.

4.1 Introduction

Garg et al. [40] investigated the performance of transformers, when trained from scratch, in solving specific function class learning problems in-context. Notably, transformers exhibited strong performance across all tasks, matching or even surpassing traditional solvers. Building on this, Akyürek et al. [3] explored the transformer's capability to learn linear regression task, interpreting the learned model as an implicit form of established learning algorithms. Their study included both theoretical and empirical perspectives to understand how transformers learn these functions. Subsequently, von Oswald et al. [141] demonstrated empirically that, when trained to predict the linear function output, a linear self-attention-only transformer inherently learns to perform a single step of gradient descent to solve the linear function in-context. While the approach and foundational theory presented by von Oswald et al. [141] are promising, there exists a significant gap between the simplified architecture they examined and the transformer used in practice. The challenge of training transformers from scratch, with only minor architectural modifications, to effectively replicate a learning algorithm remains open.

In traditional machine learning, *iterative* algorithms are commonly used to solve

linear functions. However, the methodologies employed by transformers are not naturally structured for iterative computation. A *looped* transformer architecture, as studied in Giannou et al. [47], Dehghani et al. [32], provides a promising avenue to bridge this gap. By reusing the same parameters across multiple computation steps, the looped architecture performs implicit iterative refinement of the model's latent hidden states—a form of latent computation that does not require generating any explicit tokens. This parameter reuse also means each iteration handles a simpler subtask, potentially leading to significant savings in model parameters.

To illustrate how task breakdown leads to parameter savings, consider using the transformer model to solve the linear function, specifically recovering $\boldsymbol{w}$ in $\min_{\boldsymbol{w}} \|\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y}\|_2^2$ (fig. 4.1). To train a transformer on this task, we input a prompt sequence formatted as $(\boldsymbol{x}_1, \boldsymbol{w}^\top \boldsymbol{x}_1, \ldots, \boldsymbol{x}_k, \boldsymbol{w}^\top \boldsymbol{x}_k, \boldsymbol{x}_{\text{test}})$. Here $\boldsymbol{w}$ represents the parameters of the linear function, $\{\boldsymbol{x}_1, \cdots, \boldsymbol{x}_k\}$ are k in-context samples, and $\boldsymbol{x}_{\text{test}}$ is the test sample. The transformer can potentially try to predict y_{test} by approximating the ordinary least squares solution in a single forward pass. The computation of the matrix inverse, as required in the ordinary least squares solution $(\boldsymbol{X}^\top \boldsymbol{X})^{-1} \boldsymbol{X}^\top \boldsymbol{y}$, is more difficult for transformers to learn compared to matrix multiplication [23, 141]. This is attributed to the increased number of layers and heads required in the inverse operation [47]. Nevertheless, gradient descent offers an alternative approach to solving the linear function, which requires only the matrix multiplication: $\boldsymbol{X}^\top(\boldsymbol{X}\boldsymbol{w} - \boldsymbol{y})$, but is applied repeatedly.

Motivated by this observation, we ask the following question:

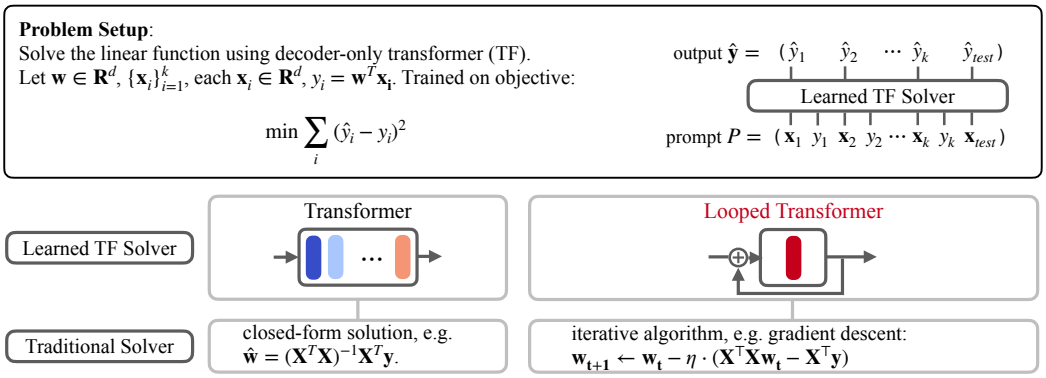


Figure 4.1: *How Can a Transformer Be Trained to Learn an Iterative Learning Algorithm?* Here we consider the task of training a transformer to solve the linear function in context. The provided prompt $(\boldsymbol{x}_1, y_1, \boldsymbol{x}_2, y_2, \cdots, \boldsymbol{x}_k, y_k, \boldsymbol{x}_{\text{test}})$ is fed into a decoder transformer. The objective is to reduce the squared loss between the predicted $\hat{y}_{\text{test}}$ based on this prompt, and the target value $f(\boldsymbol{x}_{\text{test}})$. Garg et al. [40] demonstrated that a decoder transformer can learn to solve linear functions, which potentially involves learning the approximation of the least squares solution. In this chapter, we aim to train a transformer to learn iterative learning algorithms. Our goal is to achieve performance on par with transformers but with fewer parameters. To this end, we introduce the looped transformer architecture and its accompanying training methodology.

Can looped transformers emulate iterative learning algorithms more efficiently than non-recursive transformers?

Within the specific function classes tested, the answer is positive for in-distribution evaluation. Our preliminary findings on using looped transformer to solve linear functions are illustrated in fig. 4.2. The remainder of the chapter is organized as follows. In section 4.3, we develop a training method for the looped transformer to emulate the desired performance of the iterative algorithm. Subsequently, in section 4.4, we compare the empirical performance of the transformer and the looped transformer and analyze the trade-off between them. Our contributions and findings are outlined below:

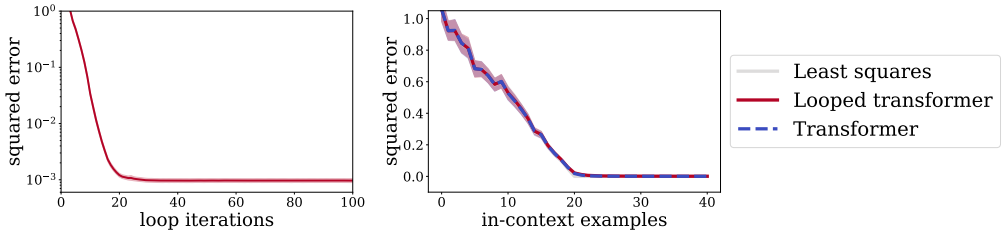


Figure 4.2: *The Looped Transformer Can Emulate Iterative Learning Algorithms, Offering Performance Comparable to Transformers with Reduced Parameters*. We train a *looped* transformer to solve linear functions in-context. (*Left*): While trained for 30 loop iterations, the looped transformer during inference achieves a stable fixed-point solution beyond the trained loop iterations. (*Right*): The trained looped transformer matches the performance of a 12-layer transformer and closely aligns with the least squares solver, while using only 1/12 of the transformer's parameters.

Training Methodology for Looped Transformer. We propose a training methodology for looped transformers, aiming to effectively emulate iterative algorithms. The assumption for a looped transformer simulating a convergent algorithm is that as loop iterations increase, the performance of the looped transformer should improve or converge. In alignment with this assumption, we delve into the structural design of the looped transformer, as well as investigate the number of loop iterations required during training. These investigations lead to the formulation of our training method.

Performance of Looped Transformers for In-Context Learning. Based on our proposed training algorithm, empirical evidence demonstrates that looped transformer can be trained from scratch to in-context learn data generated from linear functions, sparse linear functions, decision trees, and 2-layer neural networks. Among the varied function classes examined, the looped transformer consistently matches or outperforms the transformer, particularly when data is generated from

sparse linear functions or decision trees. Our findings hint at the possibility that the looped transformer is more effective at in-context emulating learning algorithms, specifically for the learning tasks explored in this chapter.

4.2 Problem Setting

Let $\mathcal{F}$ denote a class of functions defined on $\mathbb{R}^d$. Let $\mathcal{D}_{\mathcal{F}}$ denote a probability distribution over $\mathcal{F}$ and let $\mathcal{D}_{\mathcal{X}}$ denote a probability distribution on $\mathbb{R}^d$. A random learning prompt P is generated as follows. A function f is sampled from $\mathcal{D}_{\mathcal{F}}$ and inputs $\{\boldsymbol{x}_1, \cdots, \boldsymbol{x}_k\}$ as well as the test sample $\boldsymbol{x}_{\text{test}}$ are sampled from $\mathcal{D}_{\mathcal{X}}$. The output of $\boldsymbol{x}_i$ is computed by $f(\boldsymbol{x}_i)$. The prompt is then $P = (\boldsymbol{x}_1, f(\boldsymbol{x}_1), \cdots, \boldsymbol{x}_k, f(\boldsymbol{x}_k), \boldsymbol{x}_{\text{test}})$ and the goal of a learning system is to predict $f(\boldsymbol{x}_{\text{test}})$. Let M be a learning system and let $M(P)$ denote its prediction (note it is not given f explicitly). The performance of M is measured by the squared error $\ell(f(\boldsymbol{x}_{\text{test}}), M(P)) = (M(P) - f(\boldsymbol{x}_{\text{test}}))^2$. In this chapter, we focus on transformer-based learning systems and compare them with other known learning systems depending on the tasks. Specifically, we examine the GPT-2 decoder model [116] with L layers. By default, $L = 12$ for the transformer, following the setting of Garg et al. [40], and $L = 1$ for the looped transformer.

4.3 Training Algorithm for Looped Transformer

In this section, we delve into the design choice for the algorithm-emulated looped transformer. For an algorithm-emulated looped transformer, we anticipate the

following characteristics: 1) As loop iterations progress, the looped transformer should maintain or improve the performance; 2) the loop iterations have the potential to continue indefinitely without deterioration in performance.

Training Strategy. Building on the problem setting in section 4.2, let the prompt to the transformer be $P = (\boldsymbol{x}_1, f(\boldsymbol{x}_1), \cdots, \boldsymbol{x}_k, f(\boldsymbol{x}_k), \boldsymbol{x}_{\text{test}})$, with P_i denoting the prompt prefix with i in-context samples $P_i = (\boldsymbol{x}_1, f(\boldsymbol{x}_1), \cdots, \boldsymbol{x}_i, f(\boldsymbol{x}_i), \boldsymbol{x}_{i+1})$. The output of the looped transformer after t looping iterations is

$$Y_t(P_i|\theta) = \underbrace{M_\theta(M_\theta(\cdots M_\theta(}_{t \text{ iterations}} Y_0^i + P_i) + P_i) \cdots + P_i)$$

where the transformer M is parameterized by θ, and Y_0^i is a zero tensor with the same shape as P_i. Then we train the transformer by minimizing the following expected loss:

$$\min_\theta \mathbb{E}_P \left[\frac{1}{b - b_0 + 1} \sum_{t=b_0}^{b} \frac{1}{k+1} \sum_{i=0}^{k} \ell \left(f(\boldsymbol{x}_{i+1}), Y_t(P_i|\theta) \right) \right], \tag{4.1}$$

where we only measure the loss of the transformer over all prompt prefixes with loop iteration b_0 to b, with $b_0 = \max(b - T, 0)$. This truncated loss is inspired by the truncated backpropagation through time [61, 60] for computation saving.

Model Configuration and Parameter Count. For fair comparison, we use a transformer identical to the one described in Garg et al. [40], except for the number of layers. Specifically, we employ a GPT-2 model with an embedding dimension of $D = 256$ and $h = 8$ attention heads. The transformer has $L = 12$ layers, and the looped transformer has $L = 1$ layer. In terms of the number of parameters, the

transformer comprises 9.48M parameters, and the looped transformer uses 0.79M. We follow the same training strategy: train with Adam optimizer, learning rate 0.0001, no weight decay or other explicit regularization (such as gradient clip, or data augmentation).

Key Factors for Finding a Fixed-Point Solution. When deploying this looped architecture training, two key factors come into consideration: 1) the input injection in the looped architecture (section 4.3.1), and 2) the maximum loop iterations during training (section 4.3.2). The subsequent sections will illustrate the impact of these two factors, using the linear function as the specific task for in-context learning.

4.3.1 Input Injection to Looped Transformer

Finding

Input injection avoids divergence when the model is run for more iterations than seen during training.

Reusing some notation, let P represent the inputs to the transformer model M, and let Y_t be the output after applying M for t iterations. In a general form, a looped transformer can be represented as $Y_{t+1} = M(Y_t; P)$, $\forall t$. Several studies [77, 32] have investigated a specific variant termed the *weight-tying* form: $Y_{t+1} = M(Y_t)$ with $Y_0 = P$, for a finite number of iterations.

However, as t approaches infinity, the influence of the initial input P dimin-

ishes, and the solution becomes essentially random or unpredictable. Similar results have been observed in Bai et al. [7], Bansal et al. [13], where Bansal et al. [13] propose to always recall the input when training a recurrent convolutional neural network to solve the maze problem, and Bai et al. [7] show by construction that any traditional L-layer neural network can be represented by a weight-tied, input-injected network. Motivated by these findings, we propose setting $Y_{t+1} = M(Y_t + P)$ in order to incorporate the input P at each loop. It should be noted that input injection methods are not limited to addition only. In fig. 4.3, we display the outcomes when training a looped transformer either with (default) input injection or without (referred to as weight-tying). During inference, the weight-tying transformer will quickly diverge after the trained loop iterations.

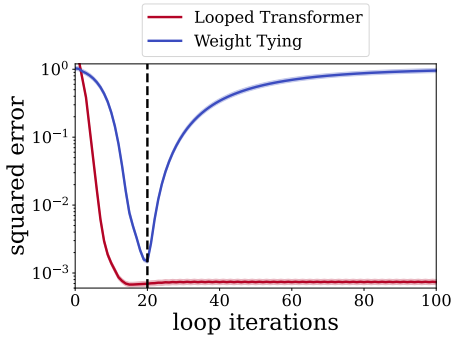


Figure 4.3: *Test Error for Linear Functions Using Looped Transformer, Comparing Models with Default Input Injection to Those Without.* Without input injection, the transformer's performance deteriorates beyond the trained loop iterations.

4.3.2 Choice of Loop Iterations

To assess the looped transformer performance at the b-th loop iteration, we must execute this computation b times consecutively. This can be analogized to a trans-

former architecture possessing b layers in effect, albeit with shared weights throughout these layers. Choosing the value of b requires a balance. A higher b leads to longer training and inference time, whereas a lower b limits the model's potential. Furthermore, as denoted in eq. (4.1), the loss is truncated, with only the outputs of T loop iterations contributing to the loss function.

Similarly, there's a trade-off when choosing T: a smaller T results in reduced memory usage but negatively impacts performance, whereas a larger T may cause the gradient to fluctuate, making the training process unstable. This section focuses on the optimal selection of b and T values (fig. 4.4) for linear functions. We analyze the suitability of b values within the set $\{12, 20, 30, 40, 50\}$, and T values in $\{5, 10, 15\}$ for the linear function.

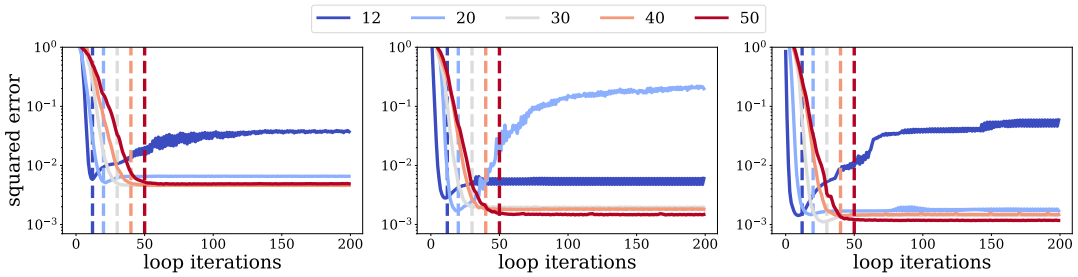


Figure 4.4: *Evaluation of the Looped Transformer on In-Context Learning Linear Functions with Different b and T During Training*. The figure from left to right is trained with $T = 5, 10, 15$, and different colors present different b values (denoted in the legend). The solid lines of various colors depict how the looped transformer, trained with a specific value of b, performs as the loop iteration increases during inference. The corresponding dashed line represents the value of b.

Finding

A sufficiently large loop depth b enables the looped transformer to find a stable fixed-point solution.

Figure 4.4 suggests that a lower b value might cause the looped transformer to diverge after the trained loop iterations, leading to a less robust fixed-point solution. On the other hand, a higher b value allows the model to locate a fixed-point solution that avoids divergence after the trained loop iterations. However, exceeding a certain b value initiates a decline in the convergence rate of the loop iteration.

A similar trade-off applies to the selection of T; a lower T may compromise performance, whereas a larger T could induce instability in training, a phenomenon documented in the recurrent model training literature [68]. A looped transformer with $b = 12$ matches the effective depth of the transformer studied in Garg et al. [40]. Nevertheless, it fails to replicate the performance of the transformer.

An additional observation is that the looped transformer consistently discovers a fixed-point solution that saturates prior to the trained iteration b. This saturation of the fixed-point solution occurs due to the loss objective, which requires the looped transformer to match the target within b steps. Consequently, selecting a smaller value of b expedites convergence to the fixed-point solution. However, beyond a certain value of b, the convergence rate reaches saturation regardless of the increase in b. For instance, training with $T = 15$, $b = 40, 50$ yields similar convergence rates.

Optimal Choice of b and T. Our goal is to efficiently train the looped transformer for the in-distribution task. This requires determining the minimal b value that prevents divergence after training loop iterations. To minimize memory usage, we prefer a smaller T value. Guided by these criteria, we adopt $b = 30$ and $T = 15$ for linear functions.

4.4 Experimental Results

4.4.1 Experimental Setup

We focus on the data-fitting problems generated by the linear model with problem dimension $d = 20$, and in-context sample $k = 40$. The parameters are sampled from $\mathcal{N}(0, \boldsymbol{I})$, and the in-context samples $\boldsymbol{x}_i \sim \mathcal{N}(0, \boldsymbol{I})$ as well. When measuring the performance, we evaluate 1280 prompts and report the 90% confidence interval over 1000 bootstrap trials. To ensure the error is invariant to the problem dimension, we also normalize the error as specified in Garg et al. [40].

Scheduled Training. We follow the training curriculum in terms of d and k, as specified in Garg et al. [40]. Additionally, we implement a schedule on the parameter b, the maximum number of loop iterations, which is progressively increased during training. The truncated loss is computed over a window of size T, covering loop iterations $\max(b - T, 0)$ to b as defined in eq. (4.1).

Consequently, we avoid referring to this approach as “curriculum learning," as that term typically refers to starting with a less difficult task and gradually moving

on to more challenging ones. Rather, this strategy can be seen as a method to warm up the training of the transformer model in a loop structure. In general, the decision to use or not to use the scheduling does not significantly impact the outcome.

4.4.2 Looped Transformer can in-context Learn Linear Functions

Finding

Looped transformer matches the performance of transformer when evaluating in-distribution.

As indicated in fig. 4.2 (right), the looped transformer trained with $b = 30$ and $T = 15$ is able to match the performance of the transformer, almost matching the performance of the traditional optimal solver: the ordinary least squares.

Finding

Looped transformer learns efficiently with fewer distinct in-distribution prompts.

We further evaluate the sample complexity of transformer training. Specifically, we ask how many distinct prompts need to be seen during training for the transformer or the looped transformer to learn the function. To do so, instead of generating the linear function on the fly in each iteration, we generate the training dataset before training. Thus, during training, the transformer may encounter the same prompt multiple times.

To isolate the impact of curriculum learning, we disable this strategy and focus on linear functions with problem dimension $d = 10$, in-context samples $k = 20$. The results are presented in fig. 4.5. Due to the fewer parameters in the looped transformer, it is able to learn the function with fewer distinct prompts/functions compared to the transformer. More results regarding different problem dimensions are presented in section 4.8.

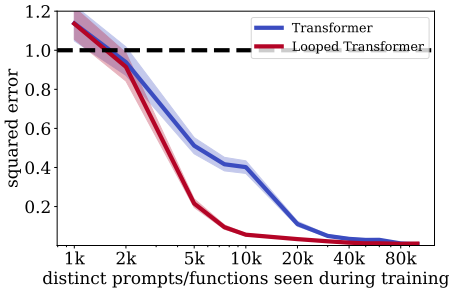
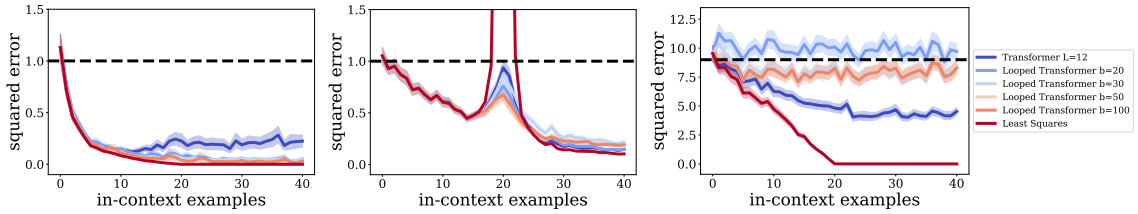


Figure 4.5: *Performance of Transformers on Linear Functions with $d = 10$ and $k = 20$, When Trained with Different Numbers of Distinct Prompts/Functions.*

Finding

Looped transformer exhibits an inductive bias toward simpler solutions when evaluating out-of-distribution.

Recall that the transformer is trained with $\boldsymbol{x} \sim \mathcal{N}(0, \boldsymbol{I})$. We now evaluate it on: a) inputs with skewed covariance, b) inputs with noise, and c) scaled in-context example inputs. The result is shown in fig. 4.6. Rather than learning the true algorithm (e.g., OLS or gradient descent) for solving the linear function class, the looped transformer learns an approximation that exhibits an inductive bias favoring simpler solutions compared to the transformer. As a result, it handles (a)



(a) Skewed Covariance (b) Noisy Linear Regression (c) Scaled x with scaling=3

Figure 4.6: *Evaluation of Transformers Trained on Linear Functions, Tested Under Various Conditions: a) Inputs with Skewed Covariance, b) Noisy Linear Functions, and c) Scaled Inputs.*

skewed covariance inputs better than the transformer.

In the task of (b) noisy linear regression, the looped transformer displays a reduced peak in the double descent curve, similar to the effects of applying minor regularization to the base model for resolving noisy linear regression. However, this inductive bias towards simplicity can negatively impact performance when there is a scaling shift in the input distribution. As depicted in fig. 4.6 (c), during the inference process, if we sample $\boldsymbol{x}$ such that $\boldsymbol{x} = 3\boldsymbol{z}$, and $\boldsymbol{z} \sim \mathcal{N}(0, \boldsymbol{I})$, the looped transformer underperforms compared to the transformer.

4.4.3 Impact of Model Architecture Variations on Looped Transformer Performance

In this section, we explore the impact of varying the number of layers (L) on the looped transformer. The experiments are trained with $b = 30$, $T = 15$.

In fig. 4.7, we plot the squared error for the transformer with L layers, and the looped transformer with L layers, applying t loop iterations. From the figure, we observe that as L increases, convergence is achieved more rapidly. To match the

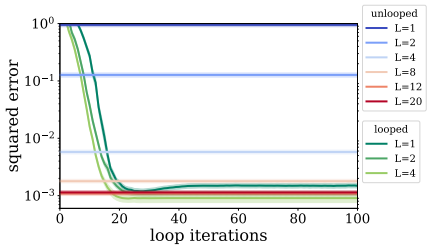


Figure 4.7: *Transformer and Looped Transformer's Performance under Different Depth* (*Loops*). For linear functions with problem dimension $d = 20$, and in-context samples $k = 40$, we test the transformer and looped transformer with $D = 256$ and $h = 8$, but varying L, where L is the number of layers.

performance of a transformer with L layers, the looped transformer needs more than L loop iterations, *i.e.*, the effective depth of the looped transformer exceeds that of the transformer. For instance, a looped transformer with a single layer requires roughly 20 iterations to achieve an 8-layer transformer's performance. This suggests that the transformer and the looped transformer learn different representations at each layer (iteration).

To delve deeper into the learning dynamics of each layer in both the transformer and the looped transformer, we employ the model probing technique suggested by Akyürek et al. [3], Alain and Bengio [5]. Reusing some notation, we represent the output of the transformer's t-th layer or the looped transformer's t loop iterations as Y_t. An MLP model is trained on Y_t to learn target probes, specifically $\boldsymbol{X}^\top \boldsymbol{y}$ for the gradient component and the $\boldsymbol{w}_{\text{OLS}}$ for the optimal least squares solution. The mean squared error for model probing is illustrated in fig. 4.8. While transformers initially capture representations relevant to $\boldsymbol{X}^\top \boldsymbol{y}$ and $\boldsymbol{w}_{\text{OLS}}$, and subsequently shift focus to other statistics, the looped transformer consistently refines its representations related to the target probes across its loop iterations. Details of

the model probing are presented in section 4.10.

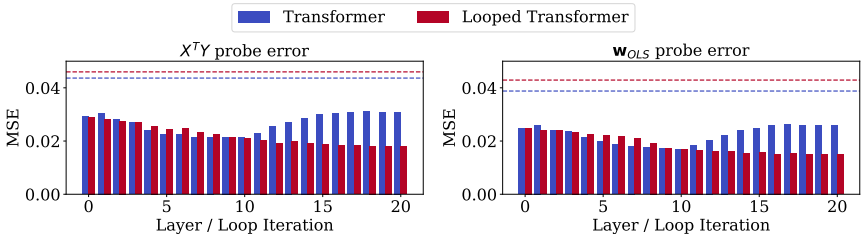


Figure 4.8: *How Do the Transformer and the Looped Transformer Encode Information Across Layers/Iterations?* We train a 2-layer MLP probing model to recover the target probe from the transformer's output at t-th layer/loop iteration. The dashed line indicates the minimal probe error obtained in a control task, where the linear function parameter $\boldsymbol{w}$ is fixed to be $\mathbf{1}$. Contrasting with the transformer, which decodes the target probe optimally around the 10-th layer, the looped transformer steadily refines its representation, improving the decoding of the target probe with each subsequent loop iteration.

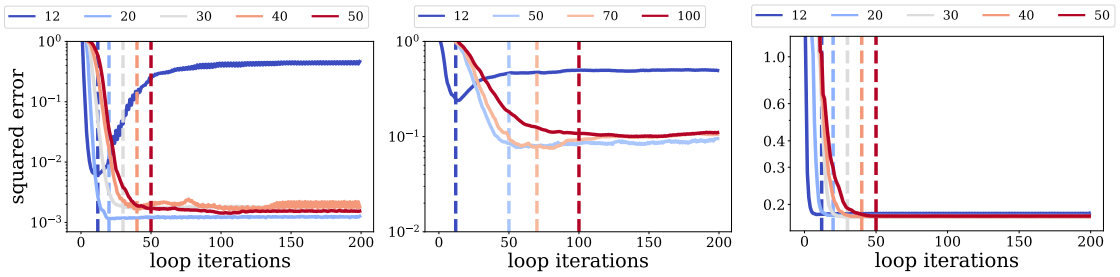
4.4.4 Higher Complexity Function Classes

In this section, we shift our focus to function classes of higher complexity: a) the sparse linear model with $d = 20$, $k = 40$, and $s = 3$ nonzero entries; b) the decision tree with depth 4, input dimension $d = 20$, and $k = 100$; c) the 2-layer ReLU neural network with 100 hidden neurons, input dimension $d = 20$, and $k = 100$. For these functions, parameters are sampled from $\mathcal{N}(0, \boldsymbol{I})$, and in-context samples $\boldsymbol{x}_i \sim \mathcal{N}(0, \boldsymbol{I})$. This setup follows the methodology described in Garg et al. [40], and is further detailed in section 4.6.

Optimal Loop Iterations in the Looped Transformer for Different Functions. In section 4.3.2, we highlight that for the looped transformer trained on linear functions, a larger value of b and an appropriate T enhance its ability to discover a

fixed-point solution that remains stable beyond the trained loop iterations. This observation is consistent across various functions, but the optimal b and T values may vary. fig. 4.9 depicts the performance of looped transformer trained with $T = 10$ but varying b values on different functions.

Comparison with fig. 4.4 indicates that tasks with higher complexity, like the linear function (which is more parameter-intensive than the sparse linear function), necessitate increased b and T values to achieve stable convergence during training. For instance, with $T = 10$, the linear function diverges at $b = 20$, whereas the sparse linear function converges to a fixed point. The values of b and T required can be indicative of the complexity a transformer faces when tackling a specific task. Interestingly, while learning a 2-layer neural network is more challenging [24], it requires fewer b and T for the looped transformer in comparison to seemingly simpler tasks, such as the linear function. Complete results of the looped transformer trained with various b and T are presented in section 4.7.



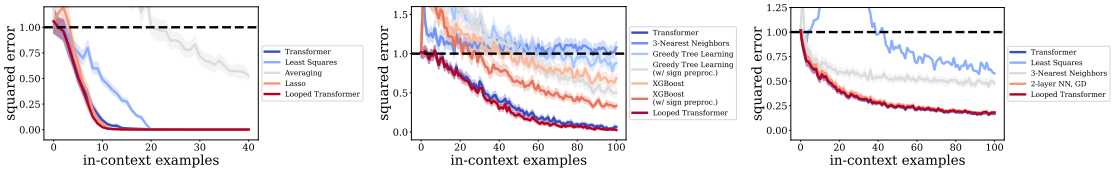
(a) Sparse Linear Function. (b) Decision Tree. (c) 2-layer ReLU NN.

Figure 4.9: *Evaluation of the Looped Transformer on In-Context Learning Data Generated from a) the Sparse Linear Function, b) the Random Decision Tree, and c) the 2-Layer ReLU Neural Network with $T = 10$ and Different b During Training.* The solid lines of various colors depict how the looped transformer, trained with a specific value of b, performs as the loop iteration increases during inference. The corresponding dashed line represents the value of b.

Finding

On higher-complexity function classes, the looped transformer matches or outperforms the transformer using less than 10% of the parameters.

Looped Transformer Matches or Outperforms Transformer. Through a comprehensive hyperparameter search, we identify optimal values for b and T, as detailed in section 4.7. Specifically, for data generated from a) sparse linear functions, we use $b = 20$ and $T = 10$; b) decision trees, $b = 70$ and $T = 15$; and c) 2-layer ReLU NNs, $b = 12$ and $T = 5$. We then compare the performance of looped transformer against the transformer and other baseline solvers, as depicted in fig. 4.10. Of all the tasks we examine, the looped transformer consistently matches the transformer, except for the sparse linear function tasks, where the looped transformer *outperforms* the transformer, and even the Lasso [138]. By *outperforming*, we do not refer to the precision of the final solution but to the trend relative to the number of in-context samples. For instance, with 40 in-context samples, the Lasso achieves a solution with an error of 1.32e-04, the Least Squares reaches 8.21e-14, the looped transformer achieves 6.12e-04, and the transformer achieves 0.0017. However, with fewer than 10 in-context samples, the looped transformer has a smaller error than the Lasso solution. We perform a hyperparameter search for Lasso over $\alpha \in \{1e-4, 1e-3, 0.01, 0.1, 1\}$, where α is the ℓ_1 parameter, and select the best $\alpha = 0.01$. We conjecture that this performance gain is a result of the inductive bias



(a) Sparse Linear Function. (b) Decision Tree. (c) 2-layer ReLU NN.

Figure 4.10: *Performance Evaluation of the Trained Transformer on In-Context Learning Data from: a) Sparse Linear Functions, b) Random Decision Trees, and c) 2-Layer ReLU Neural Networks.* The looped transformer matches or even surpasses the transformer's performance using only 1/12th of the parameters employed by the latter.

of the looped transformer, which favors simpler (*i.e.*, sparser) solutions over their more complex counterparts. When tackling sparse problems, this inherent bias enhances the performance of the looped transformer. A further analysis across varying sparsity levels is presented in section 4.9.

4.4.5 Validation on OpenML Datasets

To establish a connection with real-world applications and further validate our approach, we conduct additional experiments using 10 datasets from OpenML [140], which better represent practical scenarios. The details of the used datasets are presented in table 4.1.

Let $S = \{720, 725, 737, 803, 807, 816, 819, 847, 871, 42192\}$ be the set of all datasets. The datasets we examined have binary classification labels $(0$ or $1)$. We trained the transformer and the looped transformer on 9 datasets and evaluated the in-context learning ability on the unseen test dataset. Both transformer and looped transformer have 256 embedding size and 8 heads. The transformer has 12 layers, and the looped transformer has 1 layer, trained to maximum loop iteration $b = 30$

Dataset ID	Name	# Numerical Features	# Instances
720	abalone	7	4177
725	bank8FM	8	8192
737	space_ga	8	3107
803	delta_ailerons	5	7129
807	kin8nm	8	8192
816	puma8NH	8	8192
819	delta_elevators	6	9517
847	wind	14	6574
871	pollen	5	3848
42192	compas-two-years	7	3848

Table 4.1: *OpenML Datasets Used in the Experiments.* We only use numerical features for the input features.

and window size $T = 15$.

During training, we uniformly sampled prompts from 9 datasets, where for each prompt, we first randomly selected a training set, then randomly selected $k+1$ samples from this training set, with $k = 40$ being the number of in-context samples. During testing, we applied a similar approach for each test sample, selecting k in-context samples from the test dataset, with care taken to exclude the test sample itself from these in-context pairs. The test accuracies are presented in table 4.2. As the results indicate, the looped transformer demonstrates comparable, and in some cases, better performance compared to the transformer on these real-world datasets.

Test Dataset ID	Train Dataset IDs	Transformer	Looped Transformer
720	$S\backslash\{720\}$	0.626 ± 0.008	$\mathbf{0.662 \pm 0.008}$
725	$S\backslash\{725\}$	0.511 ± 0.007	0.504 ± 0.008
737	$S\backslash\{737\}$	0.656 ± 0.006	$\mathbf{0.720 \pm 0.010}$
803	$S\backslash\{803\}$	0.394 ± 0.010	0.400 ± 0.010
807	$S\backslash\{807\}$	0.405 ± 0.004	$\mathbf{0.416 \pm 0.005}$
816	$S\backslash\{816\}$	0.463 ± 0.004	0.462 ± 0.004
819	$S\backslash\{819\}$	0.483 ± 0.005	$\mathbf{0.568 \pm 0.010}$
847	$S\backslash\{847\}$	0.668 ± 0.007	$\mathbf{0.757 \pm 0.006}$
871	$S\backslash\{871\}$	$\mathbf{0.532 \pm 0.004}$	0.510 ± 0.005
42192	$S\backslash\{42192\}$	0.650 ± 0.005	0.650 ± 0.008

Table 4.2: *Test Accuracy for Transformer and Looped Transformer on Different Held-out Test Datasets.* The looped transformer demonstrates comparable, and in some cases, better performance compared to the transformer.

4.5 Discussion and Conclusions

Looped Transformer Emulates Fixed-Point Iteration Method Within Training Distribution. Through our designed training method, the looped transformer successfully approximates fixed-point solutions beyond the iterations it was trained on, effectively emulating the fixed-point iteration method within the training distribution. Nevertheless, these solutions come with an inherent error floor and have limited performance on out-of-distribution prompts. As a result, our trained looped transformer fails to identify an actual algorithm that generalizes to any input distribution, but only emulates the algorithm with the particular input distribution it is trained on. Overcoming this limitation, and enhancing the transformer-derived solver to match the generalization capabilities of conventional algorithms presents a promising future research direction.

Choice of b and T and the Notion of Task Difficulty. As discussed in section 4.4.4, increasing the loop iterations (b) and truncated loss window size (T) enables the looped transformer to find a fixed-point solution that does not diverge beyond the trained loop iterations. The optimal values for b and T are determined through hyperparameter tuning, selecting the minimum parameters necessary to prevent divergence in the looped transformer. These optimal values of b and T reflect the complexity of a task from the transformer's perspective. A task that is inherently more complex will require larger values of b and T to achieve convergence to a fixed-point solution, whereas simpler tasks will require smaller values. Thus, the parameters b and T could be used as a metric to assess the difficulty level of tasks for transformers. Expanding on this, in practice where tasks exhibit varying degrees of complexity, an adaptive looping strategy could be beneficial. These strategies could include token-level adaptive loop iterations, as studied in Dehghani et al. [32].

Conclusion. Motivated by the widespread application of iterative algorithms in solving data-fitting problems across various function classes, we analyzed the loop architecture choice and training strategy, then developed a training method for the looped transformer. This methodology enables the looped transformer to approximate fixed-point solutions beyond the initially trained loop iterations, using less than 10% of the parameters of a standard transformer. Our experiments show that the looped transformer achieves competitive in-distribution performance across multiple function classes while approximating fixed-point solutions beyond the trained loop iterations.

4.6 Detailed Setup for Function Classes

For completeness, we detail the setup of the function classes here, adhering to the settings in Garg et al. [40]. Inputs $\boldsymbol{x} \sim \mathcal{N}(0, \boldsymbol{I})$ are consistent across all function classes.

Linear Function. For linear function with dimension d, we sample the parameter of linear function $\boldsymbol{w} \in \mathbb{R}^d$ following $\mathcal{N}(0, \boldsymbol{I})$.

Sparse Linear Function. We follow the same distribution setting as in linear function, and we uniformly select s out of d dimension to be nonzero.

Decision Tree. We employ a binary decision tree with a depth of 4. At each non-leaf node, the decision to branch left or right is based on the sign of a randomly selected coordinate of the input $\boldsymbol{x}$. This coordinate is chosen uniformly at random. The leaf nodes are initialized following a normal distribution $\mathcal{N}(0, \boldsymbol{I})$. The evaluation of an input $\boldsymbol{x}$ involves traversing from the root to a leaf node, moving left for a negative coordinate and right for a positive one, with the output being the value of the reached leaf node.

2-layer ReLU NN. We initialize the weights of the network following $\mathcal{N}(0, \boldsymbol{I})$, and set the hidden layer dimension to be 100.

4.7 Further Analysis on Loop Iterations

In this section, we present the result (fig. 4.11) of the looped transformer performance when trained with different b and T for the following function classes: a) linear functions with problem dimension $d = 20$ and in-context samples $k = 40$, b) sparse linear functions with problem dimension $d = 20$, in-context samples $k = 40$, and nonzero entry $s = 3$, c) decision tree with problem dimension $d = 20$, depth to be 4, and in-context samples $k = 100$, as well as d) 2-layer ReLU NN with problem dimension $d = 20$, 100 hidden neurons, and in-context samples $k = 100$.

Recall that the target function we aim to minimize during the training of looped transformer is:

$$\min_{\theta} \mathbb{E}_P \left[\frac{1}{b - b_0 + 1} \sum_{t=b_0}^{b} \frac{1}{k+1} \sum_{i=0}^{k} \ell \left(f(\boldsymbol{x}_{i+1}), Y_t(P_i|\theta) \right) \right],$$

where P_i denotes the prompt prefix with i in-context samples

$$P_i = (\boldsymbol{x}_1, f(\boldsymbol{x}_1), \cdots, \boldsymbol{x}_i, f(\boldsymbol{x}_i), \boldsymbol{x}_{i+1}),$$

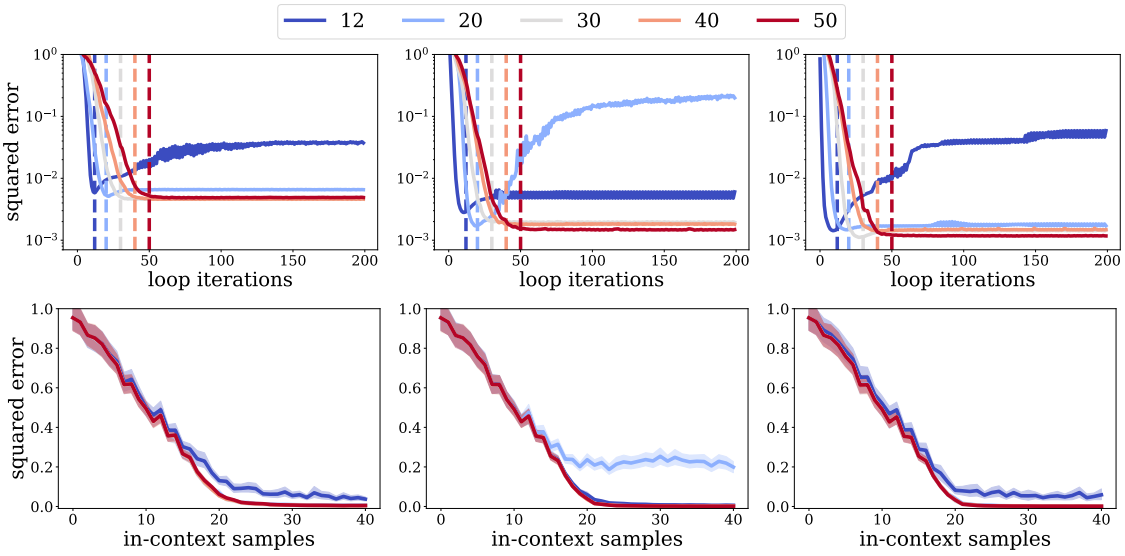
$Y_t(P_i|\theta)$ is the output of the looped transformer with parameter θ after t looping iterations. $b_0 = \max(b - T, 0)$. The choice of b affects how many loop iterations the looped transformer needs to unroll during the training, and T represents the truncated window size for the loss calculation.

For the linear function, sparse linear function, and 2-layer ReLU NN tasks, we investigate the different values of b in $\{12, 20, 30, 40, 50\}$, and for the more challenging task – decision tree, we search over b values in $\{12, 50, 70, 100\}$. For all tasks, we investigate T values in $\{5, 10, 15\}$. Larger T values such as $T = 20$ will result

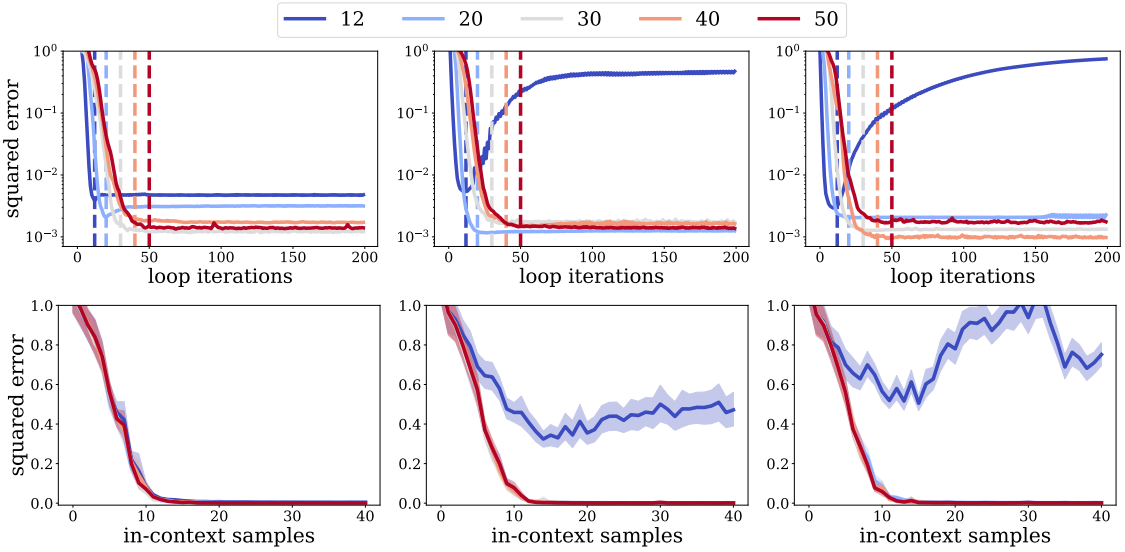
in training instability if trained without any stabilization technique such as weight decay, mixed-precision, or gradient norm clip. But once employed with those techniques, it is safe to train with large T as well. In figs. 4.11 and 4.12, we present the result of the looped transformer, when trained with different b and T, the performance on different tasks, with respect to the loop iterations, and with respect to the in-context samples.

For each task, the results are consistent: training with smaller T will yield a larger mean squared error in the in-context learning solution. When T is held constant, larger b will enable the looped transformer to find a fixed-point solution. As b increases, the performance improves until it reaches a point of saturation.

Across tasks, we can see a clear pattern of the choice of b and T related to the level of difficulty of the task. Sparse linear function, which has fewer parameters in its function class, requires fewer b and T to find a converged solution compared to the linear function. The decision tree task requires the learning of $2^{\text{depth}+1} - 1$ number of parameters (in our case, 31 parameters). Thus it requires a larger b to find a fixed-point solution. Counterintuitively, the 2-layer ReLU neural network, which contains in total 21×100 number of parameters, only requires $b = 12$ to find a fixed-point solution. The reason why data generated from a 2-layer ReLU neural network only requires a small value of b to find the fixed-point solution remains unclear. We conjecture that the value of b is an indication of the task's difficulty for the looped transformer to solve.

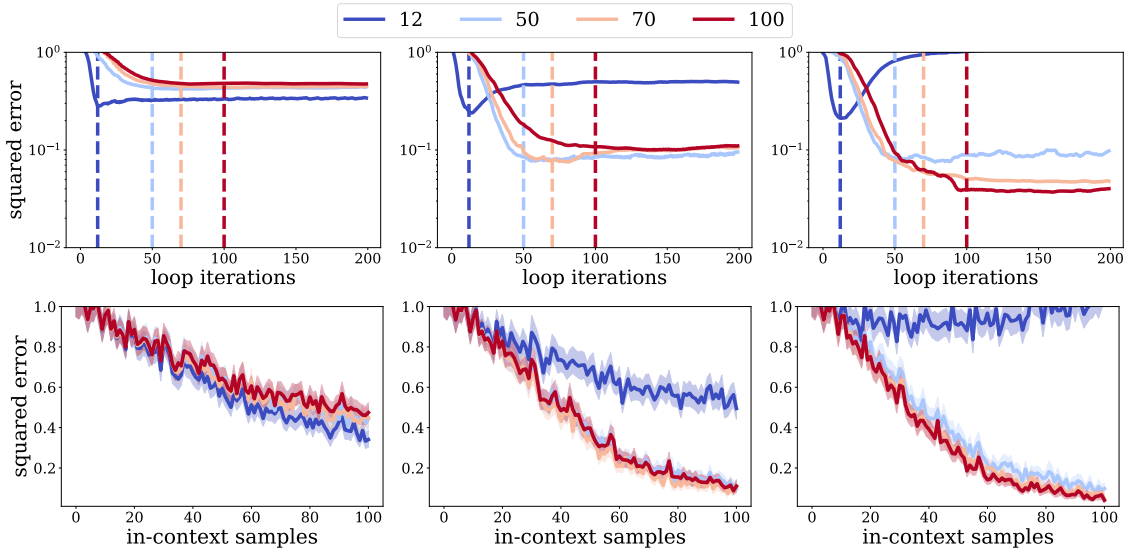


(a) Linear Function

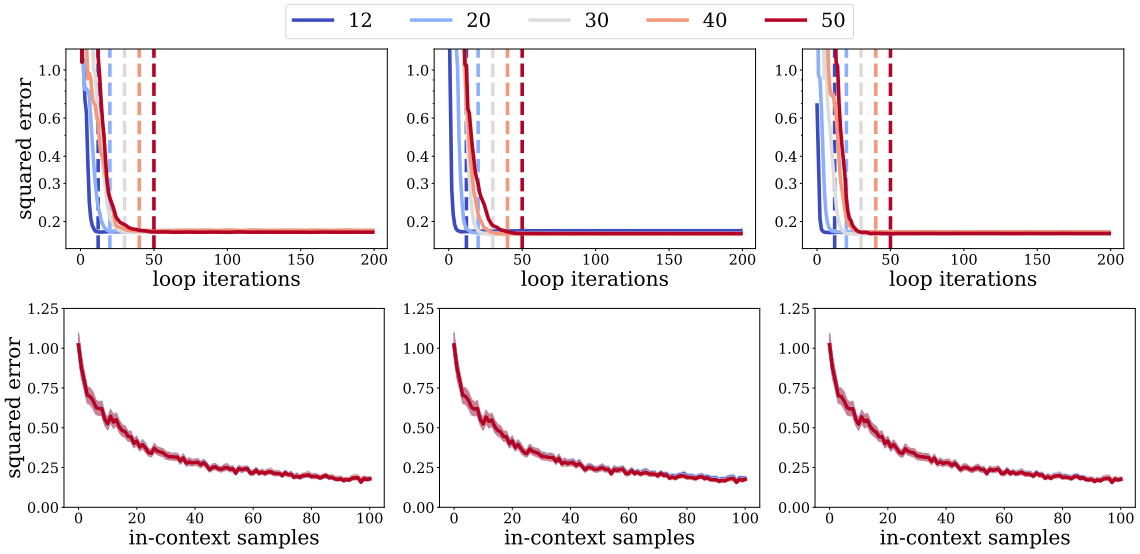


(b) Sparse Linear Function

Figure 4.11: We evaluate the looped transformer on in-context learning of a) linear functions, and b) sparse linear functions with different b and T during training (b and T is defined in eq. (4.1)). For each block of Figures, from left to right is with $T = 5, 10, 15$. (top) Performance with respect to different loop iterations, (bottom) performance with respect to different numbers of in-context samples.

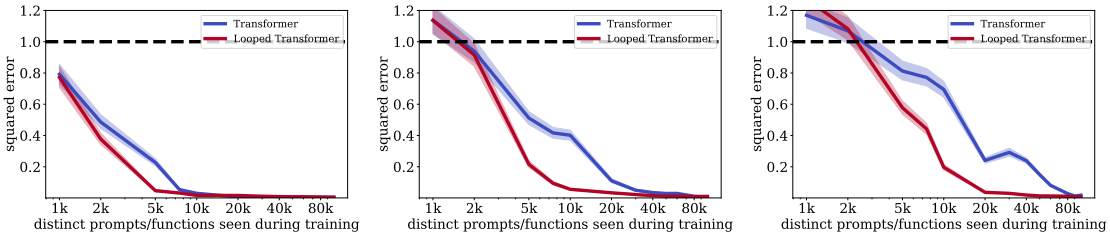


(a) Decision Tree



(b) 2-layer ReLU NN

Figure 4.12: We evaluate the looped transformer on in-context learning of a) random decision tree, and b) 2-layer ReLU neural network with different b and T during training (b and T is defined in eq. (4.1)). For each block of Figures, from left to right is with $T = 5, 10, 15$. (top) Performance with respect to different loop iterations, (bottom) performance with respect to different numbers of in-context samples.



(a) Problem dimension 5 (b) Problem dimension 10 (c) Problem dimension 20

Figure 4.13: *Performance of the Transformer and the Looped Transformer on Linear Functions with Problem Dimension $d = 5, 10, 20$, When Trained with a Different Number of Distinct Prompts/Functions.*

4.8 Further Analysis on Training Data Diversity

In section 4.4, we investigated the impact of the number of distinct prompts during transformer training for a problem dimension $d = 10$ and in-context samples $k = 20$. In this section, we extend the results to include (1) $d = 5$, $k = 10$, and (2) $d = 20$, $k = 40$. It is important to note that all experiments were conducted without the use of curriculum learning for d and k. For the looped transformer, we set $b = 30$, and $T = 15$, and we don't apply scheduling in the loop iterations as well. For $d = 5$, we investigated learning rate in $\{0.0001\}$, for $d = 10$ with learning rate $\{0.00001, 0.0001\}$, and for $d = 20$, we explored learning rate $\{0.00001, 0.0001, 0.001\}$, selecting the result with the best performance. The result is presented in fig. 4.13. Across different problem dimensions, the looped transformer demonstrated better sample efficiency compared to the transformer.

4.9 Further Analysis on Sparsity Levels

The looped transformer demonstrates enhanced performance in the sparse linear function task compared to the transformer. To further investigate this, we examine the task under varying levels of sparsity, which represent different task difficulties. To enable the looped transformer to handle all complexity levels, we choose parameters $b = 30$ and $T = 15$. As presented in fig. 4.14, the results show that across all sparsity levels of the weight vector for the sparse linear function, the looped transformer consistently outperforms the transformer, especially when the number of in-context samples is limited.

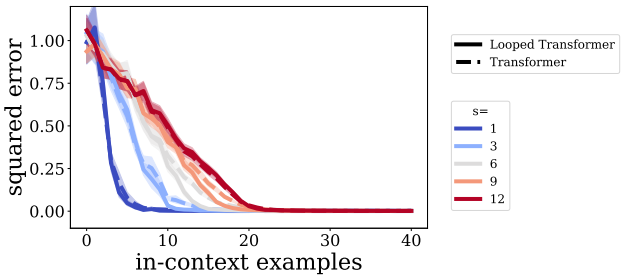


Figure 4.14: *Performance of Looped Transformer in Solving Sparse Linear Functions with Problem Dimension* $d = 20$. We examine sparsity levels $s = 1, 3, 6, 9, 12$, where only s entries are non-zero. Across all sparsity levels, the looped transformer consistently matches or outperforms the transformer, achieving more accurate solutions with fewer in-context samples.

4.10 Details of Model Probing

In section 4.4.3, we analyze the output of either the transformer's t-th layer or the looped transformer's t-th loop iteration by training a probing model on them. In this section, we present the details of the model probing experiments.

Reusing the notation, we denote the output at the t-th layer or loop iteration as Y_t. Y_t is of shape $k \times D$, where k represents the sequence length, and D represents the embedding dimension of the transformer. In our experiments, we investigate two target probes: (a) $\boldsymbol{X}^\top \boldsymbol{y}$, representing the gradient component, and $\boldsymbol{w}_{\text{OLS}}$, representing the optimal least square solution. Given the problem dimension is d, the target probes $\boldsymbol{v}$ are of shape $d \times 1$. For simplicity, we omit the batch size.

Before feeding the representation Y_t into the probing model, we first compute a weighted sum over the sequence dimension. Given $\boldsymbol{s} \in \mathbb{R}^{k \times 1}$, we perform the following operations:

$$\alpha = \text{softmax}(\boldsymbol{s}) \tag{4.2}$$

$$Z_t = \alpha^\top Y_t \tag{4.3}$$

With $Z_t \in \mathbb{R}^{1 \times D}$, we employ a multi-layer perceptron model (MLP) f_{MLP} to probe the output. The MLP model consists of two linear layers, with ReLU in between, with d_{hidden} denoting the number of hidden neurons in MLP:

$$\hat{\boldsymbol{v}} = f_{\text{MLP}}(Z_t)$$

When training the $\boldsymbol{s}$ and f_{MLP}, we aim to minimize the mean squared error between $\hat{\boldsymbol{v}}$ and $\boldsymbol{v}$. For each layer's output, and each in-context length, we train a distinct $\boldsymbol{s}$ and f_{MLP} to probe the target. When presenting the results, we compute the average loss over various context lengths. Additionally, to normalize the error, we divide the mean squared error by the problem dimension d.

In our experiments, we set $D = 256$, $d = 20$, and $d_{\text{hidden}} = 64$. We optimize the probing model with Adam optimizer and learning rate 0.001, which is found to be

the best among the set $\{0.01, 0.001, 0.0001\}$.

Control Task Configuration. To highlight the significance of the learned probing model, we follow the setting in Akyürek et al. [3], and train the probing model with the linear function weights $\boldsymbol{w}$ fixed at 1. In this case, the probing model will only learn the statistics of the input $\boldsymbol{X}$, eliminating the need for in-context learning. Then during inference, we randomly sample $\boldsymbol{w} \sim \mathcal{N}(0, \boldsymbol{I})$ and record the resulting probing error. The minimal probing error for the control tasks are shown as dashed lines in fig. 4.8. From the figure, the gap between the control task and the model probing results indicates the probing model utilizes the in-context information obtained in the output of each layer/loop iteration, suggesting that the probing error is significant.

Chapter 5

Stoic-Reasoner: Dual-Mode Transformers that Compress to Think and Decompress to Speak

The previous chapters studied how latent representations are formed and iteratively refined in controlled in-context learning settings. This chapter shifts to reasoning tasks with potentially long reasoning traces, and addresses the *compression* of explicit reasoning steps into compact latent representations.

Latent reasoning has emerged as an alternative to reasoning with natural language. It involves feeding back the last layer's hidden representation (soft token) to the input of the transformer. This idea is promising, since soft tokens have higher representation capacity compared to quantized vocabulary elements, *i.e.*, hard tokens. However, models trained with soft tokens can underperform chain-

of-thought reasoning on certain tasks. To address this, we propose a training framework for transformers called Stoic-Reasoner (Soft TOken Implicit Context Reasoner), in which the model learns to operate in two modes; one that processes the soft tokens (latent thinking mode) and one that decompresses the soft tokens into few reasoning steps using hard tokens from the vocabulary (local decoding mode). We focus on logical and math reasoning tasks, and finetune pretrained models of different sizes. Our method achieves similar or better performance, compared to supervised finetuning with chain-of-thought data across all tasks; while it requires reduced KV cache and allows sampling different reasoning traces at inference.

5.1 Introduction

Chain-of-thought (CoT) reasoning [146] has become central to modern language models, enabling them to solve complex problems by generating intermediate reasoning steps. However, these reasoning traces can be very long, increasing inference cost and KV cache usage. This has motivated research into *soft tokens*—continuous representations that are not part of the discrete vocabulary—as a way to compress reasoning traces into a more compact form [57, 25, 169, 126].

Various approaches have been proposed for constructing soft tokens, such as weighted linear combinations of vocabulary embeddings [173, 50], VQ-VAE–based compression [131], and hidden-state representations [57, 25]. We focus on the last approach, where soft tokens are last-layer hidden states. Prior methods in

this line either gradually replace CoT steps with soft tokens without explicit training [57], use auxiliary losses to distill from a teacher model [127], or train a separate encoder–decoder to guide latent reasoning [66]. However, effectiveness varies by task: latent reasoning methods perform well on graph exploration tasks but degrade on math reasoning, potentially due to compression of reasoning steps [57].

More fundamentally, soft tokens face two key limitations. First, supervision is weak and indirect—unlike hard tokens, there is no ground truth for what a soft token should represent. Second, models that reason purely in latent space produce deterministic trajectories and do not naturally support sampling, which is essential for post-training methods like GRPO [124] that generate and train on multiple reasoning paths for the same problem seed.

To address these issues, we propose Stoic-Reasoner, a training framework that addresses both challenges by having the model operate in two modes: a *latent thinking mode* that generates soft tokens, and a *local decoding mode* that decompresses each soft token into a few CoT steps using hard tokens. A central property of this approach is that soft tokens receive strong implicit supervision: at each chunk, the local decoding loss provides a step-level training signal that flows back through the soft tokens, guiding them to encode useful intermediate representations. Moreover, because the model decodes hard tokens at each step, it naturally supports sampling: stochasticity in the decoded tokens induces stochasticity in the subsequent soft tokens, enabling exploration of different reasoning paths.

Our main contributions and findings are outlined below:

Dual-Mode Training with Sampling. We train a single transformer to alternate between latent (soft token) and decoding (hard token) modes. Soft tokens receive step-level implicit supervision through the local decoding loss, and hard token generation naturally enables sampling for post-training algorithms.

Strong Performance with Reduced Compute. On logical and math reasoning benchmarks (GSM8k, ProsQA, ProntoQA), our method matches or exceeds both CoT finetuning and prior soft token methods while reducing KV cache requirements, enabling more efficient inference.

Better Generalization. Our method shows improved generalization to problems requiring longer reasoning chains than those seen during training.

5.2 Our Method

We train a single transformer to alternate between two modes: *latent thinking*, which produces soft tokens, and *local decoding*, which generates hard tokens. This compresses reasoning steps into soft tokens while still producing interpretable CoT outputs. We first describe the inference procedure, then the training algorithm. Figure 5.1 illustrates the inference procedure.

5.2.1 Data Preprocessing

We split each training example into k chunks, where k is a hyperparameter. Each chunk contains zero, one, or more CoT steps, followed by a special switch token

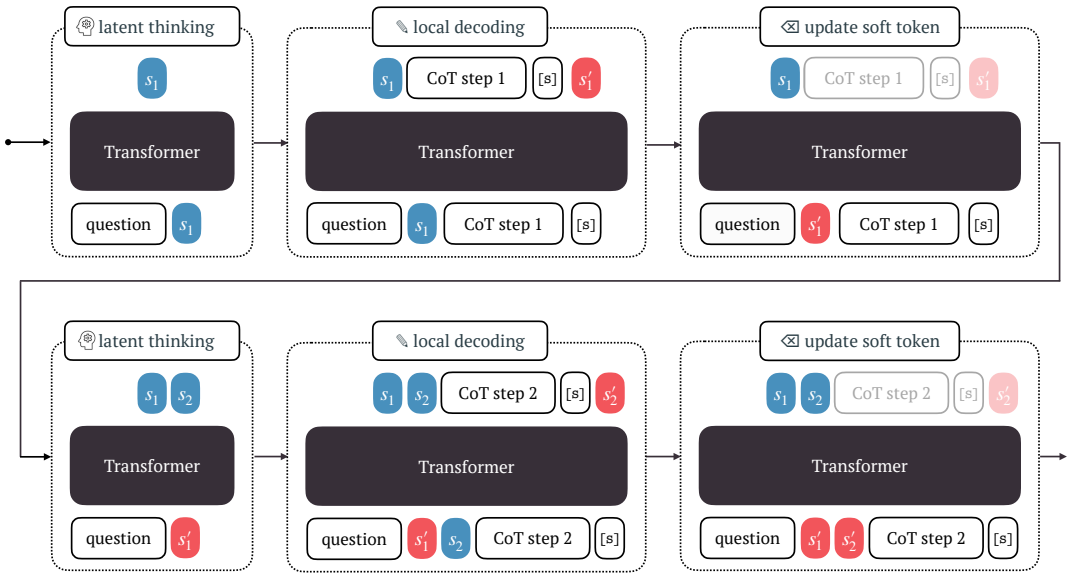


Figure 5.1: *Inference Procedure for Two Chunks.* **Top row:** Processing chunk 1. The model first produces soft token s_1 (blue) from the question alone (latent thinking). It then decodes CoT step 1, ending with switch token `[s]`. The hidden state at `[s]` becomes s_1' (red), which replaces s_1. **Bottom row:** Processing chunk 2. From question + s_1', the model produces s_2 (latent thinking). Afterwards, it decodes CoT step 2, and the hidden state at `[s]` becomes s_2', replacing s_2. The process repeats for all CoT sentence chunks that the model will generate till it produces the final answer.

`[s]`. If a CoT has fewer than k steps, extra chunks are left empty.

5.2.2 Inference

At test time, the model alternates between producing soft tokens and generating hard tokens. Below we walk through the procedure step by step. See section 5.5.1 for pseudocode.

Step 1: Initialize. Given a question, we pass it through the model and extract the hidden state at the last token position. This becomes our first soft token, s_1. It

represents the model's initial prediction of what reasoning should come next.

Step 2: Decode a Chunk. We concatenate the question and the initial guess s_1, then generate hard tokens autoregressively until the model emits the switch token `[s]`. These hard tokens form one chunk of interpretable CoT reasoning.

Step 3: Replace the Soft Token. After generating the chunk, we extract the hidden state at the `[s]` position, denoted s_1'. We replace the current soft token s_1 with s_1'. Intuitively, s_1 represents what the model *predicted* the chunk would contain, while s_1' encodes what it *actually* contained.

Step 4: Produce the Next Soft Token. We pass the question and the replaced soft tokens through the model to produce s_2. This prepares the model for the next chunk.

Step 5: Repeat. We repeat steps 2-4 for s_j with $j = 2, \ldots, k$ until the model emits `[eos]` or reaches a maximum number of chunks.

5.2.3 Training

Training follows the same alternating structure as inference (initialization, soft token replacement, and next soft token production are identical), with one key difference: instead of generating hard tokens autoregressively in Step 2, we feed the ground truth chunk tokens and compute cross-entropy loss. Specifically, we concatenate the question, all soft tokens so far, the ground truth chunk tokens, and the switch token `[s]`, then compute the standard language modeling loss on the chunk tokens. The total loss is the sum of cross-entropy losses over all chunks.

Step 1: Initialize. Same as inference. We pass the question through the model to produce the first soft token s_1.

Step 2: Compute the Loss. We concatenate the question, the initial guess s_1, the ground truth chunk tokens, and the switch token `[s]`. We feed this to the model and compute cross-entropy loss on the chunk tokens. This is standard language modeling—the model learns to predict the CoT tokens.

Step 3: Replace the Soft Token. Same as inference. We extract the hidden state at `[s]` and use it to replace s_1 with s'_1.

Step 4: Produce the Next Soft Token. Same as inference. We pass the question and the replaced soft tokens through the model to produce s_2.

Step 5: Repeat. We repeat steps 2-4 for s_j with $j = 2, \ldots, k$. The total loss is the sum of cross-entropy losses over all chunks.

Implicit Supervision. Soft tokens receive no explicit supervision. Instead, they are trained implicitly: the model must learn to produce soft tokens that enable successful decoding of subsequent chunks. Gradients flow back through the soft tokens via the cross-entropy loss on hard tokens.

5.2.4 Practical Considerations

Stochastic Decoding. Since s_j already serves as an initial prediction of the chunk content and s'_j refines it with the actual decoded tokens, we can optionally skip local decoding for some chunks. With probability p_{decode}, we decode the chunk and replace the soft token s_j with the updated s'_j. With probability $1 - p_{\text{decode}}$, we skip

decoding and keep s_j unchanged. When $p_{\text{decode}} = 0$, the model reasons entirely in latent space. This provides a tradeoff between compute and interpretability.

Soft Token Curriculum. Training jointly learns two skills: producing useful soft tokens and generating correct CoT steps. To improve stability, we use a curriculum. We start with $k = 0$ soft tokens (equivalent to CoT finetuning), then gradually increase k until reaching the target. This keeps early training close to the CoT, then transitions to soft token compression. We analyze the effect of curriculum scheduling in the experiments.

KV Cache Usage. Let q be question length, k the number of chunks, c soft tokens per chunk, and l tokens per chunk during local decoding. Our method uses KV cache scaling as $q + kc + l$. CoT with trace length L uses $q + L$. Since $kc + l \ll L$ in practice, our method substantially reduces KV cache requirements. Since the soft tokens replace the hard tokens from previous chunks, the hard tokens in each local decoding chunk attend to a shorter context, which also reduces FLOPs.

5.3 Experiments

We evaluate Stoic-Reasoner by finetuning pretrained language models on mathematical and logical reasoning benchmarks, following Hao et al. [57], Shen et al. [127]. Our main baseline throughout is supervised finetuning (SFT) on CoT traces.

Datasets. We use four datasets: GSM8k [27], iGSM [164], ProsQA [57], and ProntoQA [121]. For GSM8k we adopt the augmented split of Deng et al. [34]; for ProsQA we follow Hao et al. [57]. For iGSM we use two settings: *medium* with maximum number of operations $op \leq 15$ and out-of-distribution (OOD) evaluation at $op = 20, \cdots, 27$, and *easy* with $op \leq 9$ and OOD at $op = 12, \cdots, 19$. Dataset statistics are reported in section 5.5.2.

Models. We finetune GPT2-small (124M) [116], Gemma3 (270M) [137], and Qwen2.5 (0.5B) [115].

Training. We use identical hyperparameters for our method and the SFT baseline, first tuning for the baseline then transferring to our method. We apply learning rate 2e-4 when finetuning GPT2 and 2e-5 when finetuning Gemma3 or Qwen2.5 models. On GSM8k we finetune for 15 epochs for GPT2, and 5 epochs for Gemma3 and Qwen2.5, while for ProsQA and ProntoQA we finetune for 60 epochs. We apply weight decay 0.01, cosine annealing learning rate scheduler, and effective batch size 32. For method-specific hyperparameters, we set $p_{\text{decode}} = 0.5$ for ProsQA and $p_{\text{decode}} = 1$ elsewhere. Soft token scheduling is enabled by default, where the max number of soft tokens is set such that for the given task, each soft token encodes at most 1 CoT step. We analyze the effects of p_{decode} and the impact of scheduling in section 5.3.2.

5.3.1 Performance on Reasoning Benchmarks

Finding

On small-scale benchmarks, STOIC-REASONER matches or exceeds both CoT finetuning and prior soft token methods.

The results on different reasoning tasks and models are reported in Table 5.1. On **GSM8k**, for GPT2-small, our method outperforms the CoT baseline and other soft token approaches. On **ProntoQA**, our approach matches CoT at 100%. On **ProsQA**, latent/soft token methods generally beat CoT, likely because the task benefits from exploring multiple paths in parallel without committing to any single one. Using $p_{\text{decode}} < 1$, our method achieves significantly better performance compared to CoT and is competitive with prior soft token approaches.

Across the three backbone models, our method achieves similar or better accuracy than the CoT baseline. The gain is highest for GPT2-small, lower for Qwen2.5-0.5B, and mixed for Gemma3-270M. One possible explanation is that the benefit of soft tokens depends on vocabulary size: GPT2 uses $\sim$50k tokens, whereas Gemma3 and Qwen2.5 use larger vocabularies ($>$150k).

Finding

On iGSM, STOIC-REASONER generalizes better than CoT to problems requiring more reasoning steps than seen during training.

Model	GSM8k	ProsQA	ProntoQA
CoT	44.73	84.6	**100**
iCoT	30.0	**98.2**	–
Coconut	34.1	97.0	99.8
CODI	42.9	–	–
SbS	40.3	92.6	–
Stoic (Ours)	**47.84**	94.6	**100**

Model	CoT	Stoic (Ours)
GPT2-small	44.73	**47.84**
Gemma3-270M	**48.75**	47.92
Qwen2.5-0.5B	58.22	**58.45**

Table 5.1: **Top:** GPT2-small on GSM8k, ProsQA, ProntoQA comparing soft-token methods (results from original papers: iCoT [34], Coconut [57], CODI [127], SbS [66]). **Bottom:** GSM8k accuracy across backbone models.

Task Difficulty Generalization. We further evaluate on iGSM [164], a GSM8K-style synthetic benchmark that controls the number of arithmetic operations (op) required to solve each problem. Unlike Ye et al. [164], who train models from scratch, we finetune a pretrained GPT2-small model on iGSM-Easy ($op \leq 9$) and iGSM-Med ($op \leq 15$) datasets, and then test on out-of-distribution (OOD) data with longer reasoning chains (Easy: $op \in \{12, \cdots, 19\}$; Med: $op \in \{20, \cdots, 27\}$). Due to GPT2-small's learned absolute positional encodings, length extrapolation is limited: when CoT spans more steps than seen during training, accuracy drops for both methods. As shown in fig. 5.2, our method matches the CoT baseline on in-distribution problems and consistently retains higher accuracy on OOD problems; moreover, its accuracy decays more slowly as op increases.

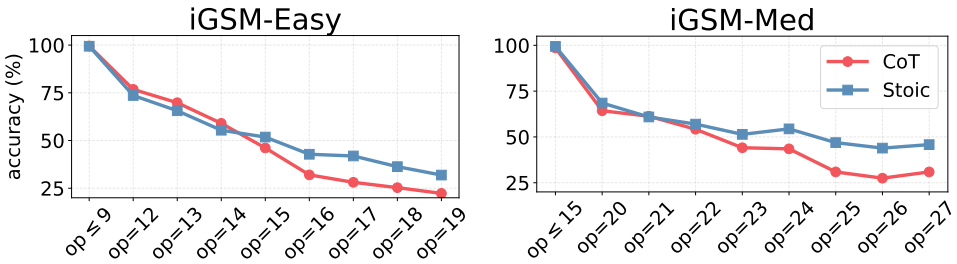


Figure 5.2: *In-Distribution and OOD Performance of iGSM Easy and Medium Dataset.* Across both settings, our method retains higher accuracy than CoT under OOD shifts with longer reasoning chains and degrades at a slower rate as the number of operations increases.

Finding

Stoic-Reasoner supports sampling via hard token generation, enabling compatibility with post-training algorithms like GRPO.

Test-Time Scaling. Stoic-Reasoner naturally supports sampling via local decoding. We evaluate `pass@K` and majority voting against the baseline. As shown in fig. 5.3, with GPT2-small trained on GSM8k, our method exhibits the same monotonic increase of accuracy with increasing K in both metrics, indicating compatibility with post-training methods. Interestingly, even though our method is slightly worse on Gemma3 under greedy decoding (Table 5.1), with temperature sampling it performs well and is more robust to higher temperatures: when the temperature T increases from 1.0 to 1.5, our method maintains similar performance, whereas the CoT baseline drops. More results on different temperatures are in section 5.6.

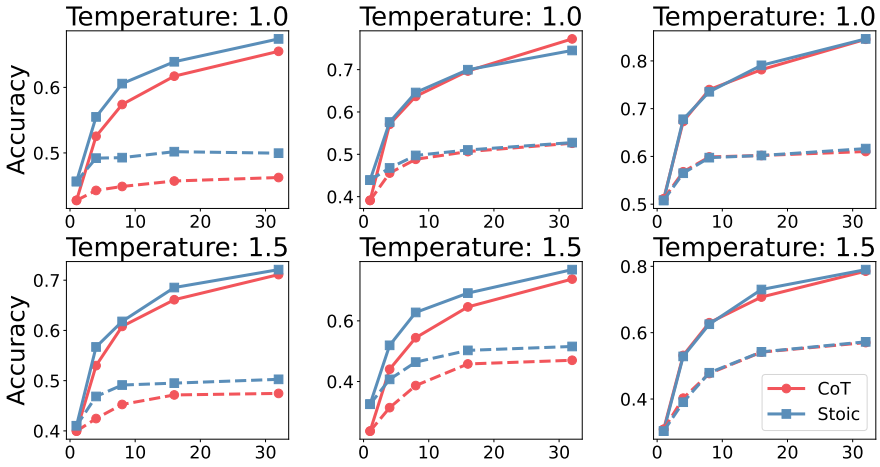


Figure 5.3: *Test-Time Scaling on GSM8k.* `Pass@K` (solid) and majority voting (dashed) for GPT2-small (left), Gemma3-270M (middle), Qwen2.5-0.5B (right). Top row: T=1.0; bottom row: T=1.5. Both methods show monotonic gains with K. Stoic-Reasoner is more robust to higher temperatures on Gemma3.

5.3.2 Ablations

Stoic-Reasoner involves the following two additional hyper-parameters compared to CoT: (i) soft token scheduling with a maximum budget $k_{\max}$, and (ii) the probability p_{decode} with which we optionally decode each soft token. In this section, we study the effects of these hyper-parameters on GSM8k and ProsQA.

Number of Soft Tokens. In GSM8k, we set the maximum number of soft tokens to $k_{\max} = 12$, since most GSM8k CoT traces contain ≤ 10 steps; this choice allows (when possible) for at most one soft token per step. To study the effects of a smaller soft token budget, we also evaluate $k_{\max} \in \{4, 6\}$, where some soft tokens must summarize more than one step on average. We report the performance of different numbers of soft tokens in Table 5.2. As shown in the table, encoding fewer steps per token improves reasoning performance, and one-per-step suffices in matching

and surpassing the performance with CoT.

	CoT	**4 soft tokens**	**6 soft tokens**	**12 soft tokens**
GSM8k	44.73	42.68	44.2	**47.84**

Table 5.2: *Performance of GPT2-Small on GSM8k for Different Number of Maximum Soft Tokens with Scheduling and Probability* $p_{decode} = 1.0$.

Scheduling on Soft Tokens. As noted in section 5.2, the model is trained to learn two skills jointly: learning the task at hand, and learning the generation of soft tokens. Introducing the full soft token budget from the start asks the model to master both skills simultaneously, which can destabilize early training. A curriculum that starts with $k = 0$ (equivalent to CoT finetuning) lets the model first learn the task at hand, then gradually introduces soft tokens so the model learns to produce useful latent representations on top of an already-grounded decoding ability. We study how different schedules for increasing k affect performance.

Specifically, we increase k by one every $N\%$ of the total training steps until $k_{\max}$ is reached; for the remaining steps of training we use $k_{\max}$. As shown in Table 5.3 , an interval of around 6% gives the best accuracy on GSM8k, balancing task learning and soft token learning. With this setting, the curriculum that gradually increases the number of soft tokens spans roughly the first 70% steps of training, while the remaining 30% steps set the number of soft tokens at $k_{\max}$, encouraging the model to compress each CoT step into a single soft token. Smaller or larger intervals still lead to performance close (or even better) to the CoT baseline. Following this rule, we use a scheduling interval of $\approx 6\%$ for both Qwen2.5 and Gemma3.

Percentage of total steps	0.0%	4.4%	5.6%	6.7%	8.3%
GPT2-GSM8k	43.44	43.9	**47.84**	46.32	47.38

Table 5.3: *Performance of GPT2-Small on GSM8k for Various Scheduling Schemes with Max Soft Token Budget* $k_{\max} = 12$. The percentage of the steps corresponds to the amount of steps performed for increasing the number of soft tokens by one. The remaining steps are performed with keeping the number of soft tokens to be the maximum. The first column (0.0%) corresponds to using the maximum number of soft tokens from the beginning.

Finding

The decoding probability p_{decode} *acts as a task-adaptive design knob: tasks that benefit from parallel latent reasoning (e.g., ProsQA) favor lower* p_{decode}*, while tasks requiring explicit grounding of intermediate steps (e.g., GSM8k) favor higher* p_{decode}*.*

Decoding Probability. During training, we stochastically decode the soft tokens via local decoding with probability p_{decode}. We study the effect of p_{decode} on ProsQA, where soft token methods outperform the CoT baseline. We sweep $p_{\text{decode}} \in \{0, 0.2, 0.5, 0.7, 1\}$, where $p_{\text{decode}} = 1$ corresponds to always using local decoding and updating the soft tokens. As shown in Table 5.4, training and evaluating with $p_{\text{decode}} = 0.5$ yields the best performance while reducing the expected number of hard-token decoding steps by 50% on average.

We also extend this analysis to GSM8k: we train with $p_{\text{decode}} = 0.5$ and vary p_{decode} at inference. As shown in Table 5.5, GSM8k accuracy drops whenever $p_{\text{decode}} < 1$, indicating that inserting latent-only steps at test time hurts performance rel-

ative to always decoding. We hypothesize this effect is task-dependent: tasks like ProsQA benefit from encoding parallel reasoning traces in the latent space, so using $p_{\text{decode}} < 1$ during training and inference can help; in contrast, GSM8k appears to benefit from grounding intermediate steps in hard tokens, so $p_{\text{decode}} < 1$ reduces accuracy, however with the trade-off of generating fewer hard tokens. These results highlight that p_{decode} can serve as a design knob that allows the same method to adapt to different types of reasoning tasks: by tuning this single parameter, one can shift the balance between latent and explicit reasoning without changing the underlying training framework.

p_{decode}	**0**	**0.2**	**0.5**	**0.7**	**1.0**
Always decoding	0.0	91.0	91.6	87.4	82.0
Probabilistic decoding	83.0	91.6	**94.6**	89.6	72.0
Latent only	90.6	89.0	94.0	93.4	0.0

Table 5.4: *Performance on ProsQA (GPT2-Small) with Variable p_{decode} (columns) and Test-Time Strategies (rows).* "Always decoding" always performs local decoding at test time; "Probabilistic decoding" uses p_{decode} matching the training column; "Latent only" uses only soft token updates. Best result is **94.6%** at $p_{\text{decode}} = 0.5$, halving hard-token decoding steps.

p_{decode}	**0**	**0.2**	**0.5**	**0.7**	**1.0**
GPT2-GSM8k	25.24	31.31	37.91	40.79	44.81

Table 5.5: *Performance of GPT2-Small on GSM8k for Variable p_{decode}.* The model is trained with $p_{\text{decode}} = 0.5$.

5.4 Discussion and Conclusions

Training Efficiency. One limitation of Stoic-Reasoner is the auto-regressive use of soft tokens. Because each subsequent soft token depends on the previous ones, training requires k sequential forward passes per example, where k is the number of chunks. With the scheduling curriculum described in section 5.2, k starts at zero and increases gradually, so early training steps are as fast as CoT finetuning; nevertheless, the overall wall-clock time scales roughly linearly with the average number of chunks. Further speedups could be achieved by approximating soft-token updates to enable parallel processing of chunks, as explored in HybridCoT [126].

Scaling and Post-Training. In this chapter, we study models up to 0.5B parameters. Scaling to billion-parameter models makes full finetuning expensive; methods such as LoRA [63] could reduce training cost, and lightweight adapters specialized for the latent-thinking and local-decoding modes could further improve their separation. On the post-training side, SFT supervises intermediate reasoning steps but provides only indirect gradients to the soft tokens through the decoding loss; reinforcement learning methods such as GRPO [124] could complement this with an outcome-based objective that directly optimizes soft token representations for final-answer correctness.

Conclusion. We presented Stoic-Reasoner, a dual-mode training framework that compresses chain-of-thought reasoning into soft tokens while preserving the ability to decode interpretable reasoning steps. By alternating between latent think-

ing and local decoding, the model learns to produce useful soft tokens implicitly through the decoding loss, without requiring explicit supervision on the latent representations. Our experiments on mathematical and logical reasoning benchmarks show that this approach matches or exceeds CoT finetuning while reducing KV cache usage, generalizes better to longer reasoning chains, and naturally supports sampling for compatibility with post-training methods.

5.5 Additional Details

5.5.1 Complete Algorithms for Training & Inference

Here we present the detailed training and inference algorithms in Algorithm 1 and 2.

Algorithm 1 Training

Input: data point (q, CoT), transformer $\texttt{TF}_\theta$, probability p_{decode}.
Choose k ▷ number of soft tokens
$\{\text{CoT}_j\}_{j=1}^{k} \leftarrow \textsc{Partition}(\text{CoT}, k)$ ▷ split CoTs into k parts
$s_1 \leftarrow \texttt{TF}_\theta(q)[-1]$
for $j = 1, \ldots, k$ **do**
    $h_j \leftarrow s_j$
    $z_j \leftarrow \texttt{TF}_\theta\big([q, h_{1:j}, \text{CoT}_j, \texttt{[s]}]\big)$
    $s'_j \leftarrow z_j[-1]$
    $t_j \leftarrow \textsc{Proj}_{\text{vocab}}(z_j[\texttt{len}(q) + j - 1 : -1])$
    if $\mathcal{U}(0,1) < p_{\text{decode}}$ **then**
        $h_j \leftarrow s'_j$ ▷ update soft token
    end if
    $s_{j+1} \leftarrow \texttt{TF}_\theta\big([q, h_{1:j}]\big)[-1]$
end for
$\mathcal{L} \leftarrow \sum_{j=1}^{k} \ell\big([\text{CoT}_j, \texttt{[s]}], t_j\big)$

Algorithm 2 Inference

Input: data point q, transformer $\texttt{TF}_\theta$, probability p_{decode}, max number of soft tokens $\max_k$.
$j \leftarrow 1$
$s_1 \leftarrow \texttt{TF}_\theta(q)[-1]$
$h_1 \leftarrow s_1$
while $j < \max_k$ **do**
  if $\mathcal{U}(0,1) < p_{\text{decode}}$ **then**
    $\mathrm{T} \leftarrow []$
    while true do
      $z \leftarrow \texttt{TF}_\theta([q, h_{1:j}, \mathrm{T}])[-1]$
      $t \leftarrow \textsc{Proj}_{\text{vocab}}(z)$
      $\mathrm{T}.\text{append}(t)$
      if $t =$ `[s]` **or** $t =$ `[eos]` **then**
        break
      end if
    end while
    if $t =$ `[eos]` **then**
      break ▷ stop generation
    end if
    $s'_j \leftarrow \texttt{TF}_\theta([q, h_{1:j}, \mathrm{T}])[-1]$ ▷ pass `[s]` through the model to obtain s'_j
    $h_j \leftarrow s'_j$ ▷ update soft token
  end if
  $s_{j+1} \leftarrow \texttt{TF}_\theta\big([q, h_{1:j}]\big)[-1]$
  $j \leftarrow j + 1$
  $h_j \leftarrow s_j$
end while

5.5.2 Dataset Details

For GSM8K, ProntoQA, and ProsQA, we adopt the preprocessing and data splits of Hao et al. [57]. For iGSM, we generate the dataset using the official open-source implementation of [164]. We record the data split in table 5.6, and the max number of CoT steps in each dataset in table 5.7. For the OOD dataset in iGSM, the number of CoT step is the number of the operations op, so the iGSM with $op = 20$ contains 20 CoT steps.

Dataset	Training	Validation	Test	OOD
GSM8k	385,620	500	1,319	-
ProntoQA	9,000	200	800	-
ProsQA	17,886	300	500	-
iGSM-med/easy	1,498,500	500	1,000	1,000

Table 5.6: *Dataset Splits.*

Dataset	Training	Validation	Test	Example CoT step
GSM8k	13	8	8	`«12+3=15»`
ProntoQA	11	11	11	`Each yumpus is a wumpus.`
ProsQA	6	6	6	`Every hilpus is a numpus.`
iGSM-med	15	13	13	`Define Niagara Falls Aviary's Enclosure as y; so y = b = 20.`
iGSM-easy	9	9	9	`Define Goat Cheese's Rye as S; so S = 3.`

Table 5.7: *Number of Maximum CoT Steps for Each Split of the Datasets.*

5.6 Additional Experimental Results

Test-Time Performance. In section 5.3.1, we study the test time evaluation performance of our method and the baseline CoT on `pass@K` and majority vote. Here we present the full results on varying temperature, completing the results demonstrated in the main text. As shown in fig. 5.4, our method exhibits similar trend in both metrics. Interestingly, on Gemma3 model, even though our method is slightly worse under greedy decoding, it is more robust to higher temperatures: as the temperature increases, our method degrades more slowly than CoT on both `pass@K` and majority voting.

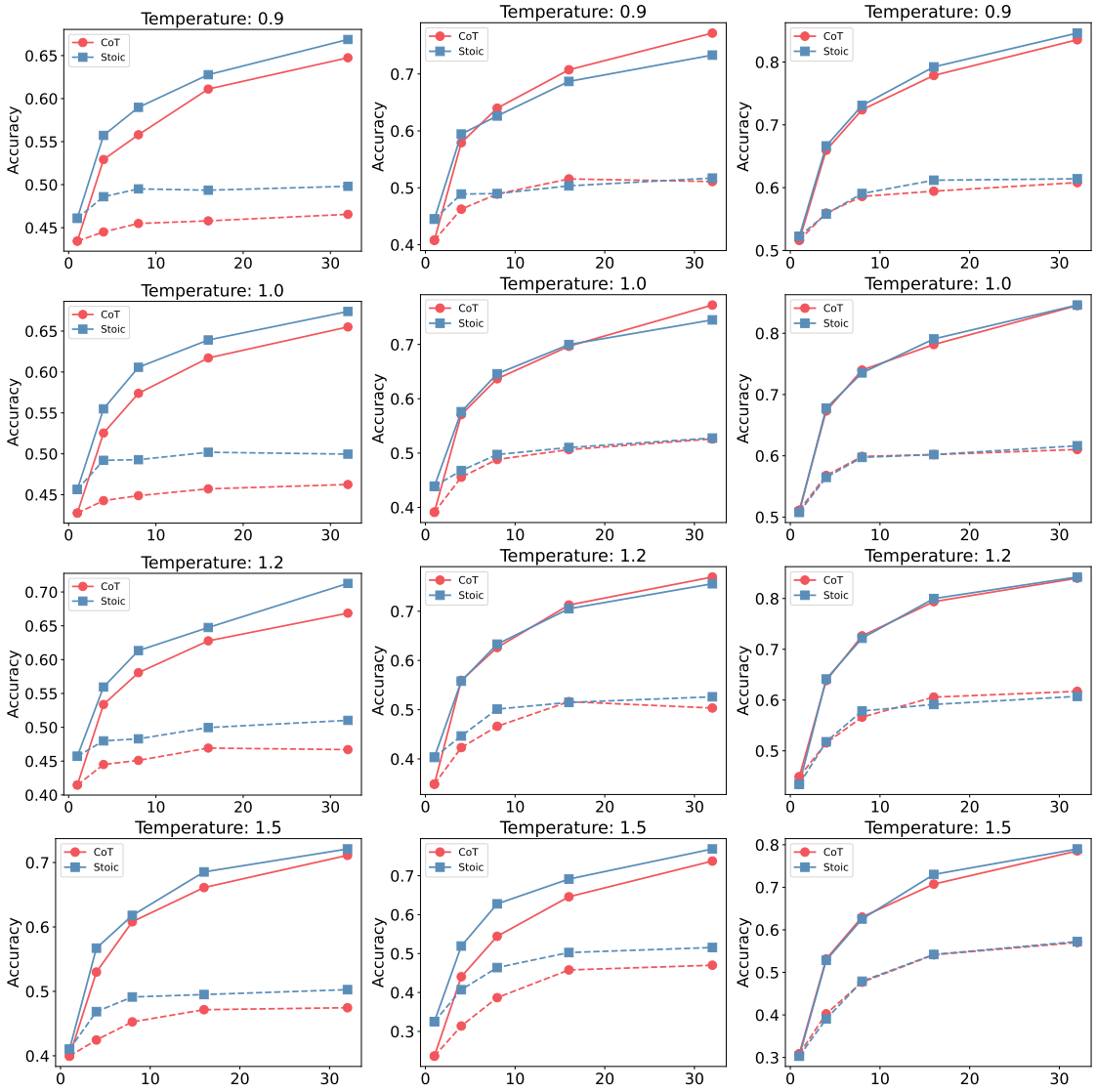


Figure 5.4: *Additional Results for GPT2-Small, Gemma3-270m and Qwen2.5-0.5B on Majority Voting and* `Pass@K`*, Using Different Temperatures.* As K increases, both methods follow a similar curve.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

Transformer models have demonstrated strong performance on a wide range of tasks by increasing inference-time computation, most notably through in-context learning and chain-of-thought reasoning. While effective, these approaches often express computation through long token sequences, leading to substantial computational and memory costs. This dissertation examined an alternative perspective, referred to as *latent thinking*, which focuses on how inference-time computation can be carried out and organized within latent representations rather than externalized entirely through tokens.

Through three complementary studies, this dissertation investigated how latent representations can be formed, iterated, and compressed to support efficient inference-time computation in transformer models. First, we analyzed how task-

dependent latent representations emerge during in-context learning and showed that encouraging their formation improves out-of-distribution robustness while maintaining in-context learning performance. Second, we studied architectural recurrence as a mechanism for iterative latent computation, demonstrating that looped transformer architectures can emulate iterative algorithms while achieving competitive performance with substantially fewer parameters. Third, we proposed a training framework for compressing explicit reasoning chains into latent representations, enabling models to reduce inference-time computation and memory usage while preserving reasoning performance.

Taken together, these results suggest that efficiency can be strongly influenced by how inference-time computation is structured and expressed. By shaping architectural design and training objectives to better leverage latent representations, it is possible to achieve efficient task adaptation and reasoning without relying on long explicit token sequences.

6.2 Limitations and Future Directions

Extending Beyond Controlled Settings to Broader Domains. Much of the analysis in this dissertation was conducted in controlled synthetic settings. The looped transformer analysis is limited to regression-based in-context learning, though concurrent works have begun exploring weight-sharing on real-world datasets [122, 44, 177] and have demonstrated promising results. For task vectors, although we verified TVP-loss on a pretrained 774M-parameter model, extending to tasks

where the underlying function is harder to describe explicitly (e.g., compositional or cognitive reasoning) would better evaluate task vector formation and reuse. For Stoic-Reasoner, scaling beyond the small models (GPT2-small, Gemma3, Qwen2.5) studied here to larger models with longer, more diverse reasoning traces [51] remains an important direction.

Adaptive Computation for Latent Thinking. A common limitation across the studies is the use of fixed computation budgets: the looped transformer uses a predetermined number of loop iterations, and Stoic-Reasoner allocates a fixed number of soft tokens per chunk. Since tasks and reasoning steps vary in difficulty, adaptive allocation would improve efficiency—assigning more loop iterations to harder inputs, or more soft tokens to complex reasoning steps. Developing principled methods for such adaptive computation, whether through learned halting mechanisms or confidence-based criteria, is a natural next step. RL is a promising direction here, since discrete decisions such as when to stop iterating or how many soft tokens to allocate are difficult to supervise but straightforward to optimize with a reward signal that balances accuracy against computation cost.

Combining Latent Thinking Across Different Axes. This dissertation studied the formation, iteration, and compression of latent representations as largely independent mechanisms. A natural direction is to explore their combination. For instance, the task vector analysis shows that task-relevant information can concentrate at specific locations within the representation, while the looped transformer iterates over the entire hidden state. An architecture that iteratively refines only

a compact task representation through looping would reduce the cost of recurrence while preserving its benefits. Likewise, combining architectural recurrence with soft token compression would allow a model to both iterate on latent states and compress intermediate reasoning, reducing KV cache usage while enabling inference-time computation through recurrence in the latent space.

Bibliography

[1] M. Aghajohari, K. Chitsaz, A. Kazemnejad, S. Chandar, A. Sordoni, A. Courville, and S. Reddy. The markovian thinker: Architecture-agnostic linear scaling of reasoning. *arXiv preprint arXiv:2510.06557*, 2025.

[2] K. Ahn, X. Cheng, H. Daneshmand, and S. Sra. Transformers learn to implement preconditioned gradient descent for in-context learning. *ArXiv*, abs/2306.00297, 2023. URL https://api.semanticscholar.org/CorpusID:258999480.

[3] E. Akyürek, D. Schuurmans, J. Andreas, T. Ma, and D. Zhou. What learning algorithm is in-context learning? investigations with linear models. *ArXiv*, abs/2211.15661, 2022. URL https://api.semanticscholar.org/CorpusID:254043800.

[4] E. Akyürek, B. Wang, Y. Kim, and J. Andreas. In-context language learning: Architectures and algorithms. *ArXiv*, abs/2401.12973, 2024. URL https://api.semanticscholar.org/CorpusID:267095070.

[5] G. Alain and Y. Bengio. Understanding intermediate layers using linear classifier probes. *ArXiv*, abs/1610.01644, 2016. URL https://api.semanticscholar.org/CorpusID:9794990.

[6] S. Bai, J. Z. Kolter, and V. Koltun. Trellis networks for sequence modeling. *ArXiv*, abs/1810.06682, 2018.

[7] S. Bai, J. Z. Kolter, and V. Koltun. Deep equilibrium models. *ArXiv*, abs/1909.01377, 2019.

[8] S. Bai, V. Koltun, and J. Z. Kolter. Multiscale deep equilibrium models. *ArXiv*, abs/2006.08656, 2020.

[9] S. Bai, V. Koltun, and J. Z. Kolter. Stabilizing equilibrium models by jacobian regularization. In *International Conference on Machine Learning*, 2021.

[10] S. Bai, V. Koltun, and J. Z. Kolter. Neural deep equilibrium solvers. In *International Conference on Learning Representations*, 2022.

[11] Y. Bai, F. Chen, H. Wang, C. Xiong, and S. Mei. Transformers as statisticians: Provable in-context learning with in-context algorithm selection. *ArXiv*, abs/2306.04637, 2023. URL https://api.semanticscholar.org/CorpusID:259095794.

[12] Y. Bai, H. Huang, C. S.-D. Piano, M.-A. Rondeau, S. Chen, Y. Gao, and J. C. K. Cheung. Identifying and analyzing task-encoding tokens in large language models. 2024. URL https://api.semanticscholar.org/CorpusID:267068913.

[13] A. Bansal, A. Schwarzschild, E. Borgnia, Z. A. S. Emam, F. Huang, M. Goldblum, and T. Goldstein. End-to-end algorithm synthesis with recurrent networks: Logical extrapolation without overthinking. *ArXiv*, abs/2202.05826, 2022.

[14] R. Belanec, S. Ostermann, I. Srba, and M. Bieliková. Task prompt vectors: Effective initialization through multi-task soft-prompt transfer. *ArXiv*, abs/2408.01119, 2024. URL https://api.semanticscholar.org/CorpusID:271693681.

[15] N. Belrose, Z. Furman, L. Smith, D. Halawi, I. V. Ostrovsky, L. McKinney, S. Biderman, and J. Steinhardt. Eliciting latent predictions from trans-

formers with the tuned lens. *ArXiv*, abs/2303.08112, 2023. URL https://api.semanticscholar.org/CorpusID:257504984.

[16] S. Bhattamishra, A. Patel, P. Blunsom, and V. Kanade. Understanding in-context learning in transformers and llms by learning to learn discrete functions. *ArXiv*, abs/2310.03016, 2023. URL https://api.semanticscholar.org/CorpusID:263620583.

[17] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. J. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Language models are few-shot learners. *ArXiv*, abs/2005.14165, 2020.

[18] A. Bulatov, Y. Kuratov, and M. Burtsev. Recurrent memory transformer. *Advances in Neural Information Processing Systems*, 35:11079–11091, 2022.

[19] A. Bulatov, Y. Kuratov, Y. Kapushev, and M. S. Burtsev. Scaling transformer to 1m tokens and beyond with rmt. *arXiv preprint arXiv:2304.11062*, 2023.

[20] N. Butt, A. Kwiatkowski, I. Labiad, J. Kempe, and Y. Ollivier. Soft tokens, hard truths. *arXiv preprint arXiv:2509.19170*, 2025.

[21] S. C. Y. Chan, I. Dasgupta, J. Kim, D. Kumaran, A. K. Lampinen, and F. Hill. Transformers generalize differently from information stored in context vs in weights. *ArXiv*, abs/2210.05675, 2022. URL https://api.semanticscholar.org/CorpusID:252846775.

[22] S. C. Y. Chan, A. Santoro, A. K. Lampinen, J. X. Wang, A. K. Singh, P. H. Richemond, J. Mcclelland, and F. Hill. Data distributional properties drive emergent in-context learning in transformers. *ArXiv*, abs/2205.05055, 2022. URL https://api.semanticscholar.org/CorpusID:248665718.

[23] F. Charton. Linear algebra with transformers. *Trans. Mach. Learn. Res.*, 2022, 2021.

[24] S. Chen, A. Gollakota, A. R. Klivans, and R. Meka. Hardness of noise-free learning for two-hidden-layer neural networks. *ArXiv*, abs/2202.05258, 2022. URL https://api.semanticscholar.org/CorpusID:246706042.

[25] J. Cheng and B. V. Durme. Compressed Chain of Thought: Efficient Reasoning Through Dense Representations, Dec. 2024. URL http://arxiv.org/abs/2412.13171. arXiv:2412.13171 [cs].

[26] A. Chevalier, A. Wettig, A. Ajith, and D. Chen. Adapting language models to compress contexts. *arXiv preprint arXiv:2305.14788*, 2023.

[27] K. Cobbe, V. Kosaraju, M. Bavarian, M. Chen, H. Jun, L. Kaiser, M. Plappert, J. Tworek, J. Hilton, R. Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

[28] D. Dai, Y. Sun, L. Dong, Y. Hao, S. Ma, Z. Sui, and F. Wei. Why can gpt learn in-context? language models implicitly perform gradient descent as meta-optimizers. 2022. URL https://api.semanticscholar.org/CorpusID:258686544.

[29] Z. Dai, Z. Yang, Y. Yang, J. Carbonell, Q. V. Le, and R. Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. *arXiv preprint arXiv:1901.02860*, 2019.

[30] G. Dar, M. Geva, A. Gupta, and J. Berant. Analyzing transformers in embedding space. *arXiv preprint arXiv:2209.02535*, 2022.

[31] J. Decugis, M. Emerling, A. Ganesh, A. Y. Tsai, and L. E. Ghaoui. On the abilities of mathematical extrapolation with implicit models. 2022.

[32] M. Dehghani, S. Gouws, O. Vinyals, J. Uszkoreit, and L. Kaiser. Universal transformers. *ArXiv*, abs/1807.03819, 2018.

[33] Y. Deng, K. Prasad, R. Fernandez, P. Smolensky, V. Chaudhary, and S. Shieber. Implicit Chain of Thought Reasoning via Knowledge Distillation, Nov. 2023. URL http://arxiv.org/abs/2311.01460. arXiv:2311.01460 [cs].

[34] Y. Deng, Y. Choi, and S. Shieber. From Explicit CoT to Implicit CoT: Learning to Internalize CoT Step by Step, May 2024. URL http://arxiv.org/abs/2405.14838. arXiv:2405.14838 [cs].

[35] N. Elhage, N. Nanda, C. Olsson, T. Henighan, N. Joseph, B. Mann, A. Askell, Y. Bai, A. Chen, T. Conerly, N. DasSarma, D. Drain, D. Ganguli, Z. Hatfield-Dodds, D. Hernandez, A. Jones, J. Kernion, L. Lovitt, K. Ndousse, D. Amodei, T. Brown, J. Clark, J. Kaplan, S. McCandlish, and C. Olah. A mathematical framework for transformer circuits. *Transformer Circuits Thread*, 2021. https://transformer-circuits.pub/2021/framework/index.html.

[36] E. Elmoznino, T. Marty, T. Kasetty, L. Gagnon, S. Mittal, M. Fathi, D. Sridhar, and G. Lajoie. In-context learning and occam's razor. *arXiv preprint arXiv:2410.14086*, 2024.

[37] Y. Fan, Y. Du, K. Ramchandran, and K. Lee. Looped Transformers for Length Generalization, Apr. 2025. URL http://arxiv.org/abs/2409.15647. arXiv:2409.15647 [cs].

[38] D. Fu, T.-Q. Chen, R. Jia, and V. Sharan. Transformers learn higher-order optimization methods for in-context learning: A study with linear models. *arXiv preprint arXiv:2310.17086*, 2023.

[39] Y. Gao, C. Zheng, E. Xie, H. Shi, T. Hu, Y. Li, M. Ng, Z. Li, and Z. Liu. Algoformer: An efficient transformer framework with algorithmic structures. *Transactions on Machine Learning Research*, 2024.

[40] S. Garg, D. Tsipras, P. Liang, and G. Valiant. What can transformers learn in-context? a case study of simple function classes. *ArXiv*, abs/2208.01066, 2022.

[41] M. Garnelo, D. Rosenbaum, C. J. Maddison, T. Ramalho, D. Saxton, M. Shanahan, Y. W. Teh, D. J. Rezende, and S. M. A. Eslami. Conditional neural processes. *ArXiv*, abs/1807.01613, 2018. URL https://api.semanticscholar.org/CorpusID:49574993.

[42] K. Gatmiry, N. Saunshi, S. J. Reddi, S. Jegelka, and S. Kumar. Can looped transformers learn to implement multi-step gradient descent for in-context learning? *arXiv preprint arXiv:2410.08292*, 2024.

[43] K. Gatmiry, N. Saunshi, S. J. Reddi, S. Jegelka, and S. Kumar. On the role of depth and looping for in-context learning with task diversity. *arXiv preprint arXiv:2410.21698*, 2024.

[44] J. Geiping, S. McLeish, N. Jain, J. Kirchenbauer, S. Singh, B. R. Bartoldson, B. Kailkhura, A. Bhatele, and T. Goldstein. Scaling up test-time compute with latent reasoning: A recurrent depth approach. *arXiv preprint arXiv:2502.05171*, 2025.

[45] M. Geva, A. Caciularu, K. Wang, and Y. Goldberg. Transformer feed-forward layers build predictions by promoting concepts in the vocabulary space. *ArXiv*, abs/2203.14680, 2022. URL https://api.semanticscholar.org/CorpusID:247762385.

[46] A. Giannou, L. Yang, K. Lee, R. D. Nowak, and D. Papailiopoulos. Stoic reasoner: Dual-mode transformers that compress to think and decompress to speak. In *The 5th Workshop on Mathematical Reasoning and AI at NeurIPS 2025*.

[47] A. Giannou, S. Rajput, J. yong Sohn, K. Lee, J. D. Lee, and D. Papailiopoulos. Looped transformers as programmable computers. *ArXiv*, abs/2301.13196, 2023.

[48] A. Giannou, L. Yang, T. Wang, D. Papailiopoulos, and J. D. Lee. How well can transformers emulate in-context newton's method? *arXiv preprint arXiv:2403.03183*, 2024.

[49] S. Goyal, Z. Ji, A. S. Rawat, A. K. Menon, S. Kumar, and V. Nagarajan. Think before you speak: Training language models with pause tokens, 2024. URL http://arxiv.org/abs/2310.02226.

[50] H. A. Gozeten, M. E. Ildiz, X. Zhang, H. Harutyunyan, A. S. Rawat, and S. Oymak. Continuous chain of thought enables parallel exploration and reasoning, 2025. URL http://arxiv.org/abs/2505.23648.

[51] E. Guha, R. Marten, S. Keh, N. Raoof, G. Smyrnis, H. Bansal, M. Nezhurina, J. Mercat, T. Vu, Z. Sprague, et al. Openthoughts: Data recipes for reasoning models. *arXiv preprint arXiv:2506.04178*, 2025.

[52] D. Guo, D. Yang, H. Zhang, J. Song, R. Zhang, R. Xu, Q. Zhu, S. Ma, P. Wang, X. Bi, et al. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948*, 2025.

[53] T. Guo, W. Hu, S. Mei, H. Wang, C. Xiong, S. Savarese, and Y. Bai. How do transformers learn in-context beyond simple functions? a case study on learning with representations. *arXiv preprint arXiv:2310.10616*, 2023.

[54] W. Gurnee and M. Tegmark. Language models represent space and time. *ArXiv*, abs/2310.02207, 2023. URL https://api.semanticscholar.org/CorpusID:263608756.

[55] S. Han, J. Song, J. Gore, and P. Agrawal. Emergence of abstractions: Concept encoding and decoding mechanism for in-context learning in transformers. 2024. URL https://api.semanticscholar.org/CorpusID:274789679.

[56] M. Hanna, S. Pezzelle, and Y. Belinkov. Have faith in faithfulness: Going beyond circuit overlap when finding model mechanisms. *ArXiv*,

abs/2403.17806, 2024. URL https://api.semanticscholar.org/CorpusID:268691935.

[57] S. Hao, S. Sukhbaatar, D. Su, X. Li, Z. Hu, J. Weston, and Y. Tian. Training Large Language Models to Reason in a Continuous Latent Space, Dec. 2024. URL http://arxiv.org/abs/2412.06769. arXiv:2412.06769 [cs].

[58] Y. He, H. S. Zheng, Y. Tay, J. Gupta, Y. Du, V. Aribandi, Z. Zhao, Y. Li, Z. Chen, D. Metzler, H.-T. Cheng, and E. H. Chi. Hyperprompt: Prompt-based task-conditioning of transformers. In *International Conference on Machine Learning*, 2022. URL https://api.semanticscholar.org/CorpusID:247218062.

[59] R. Hendel, M. Geva, and A. Globerson. In-context learning creates task vectors. *ArXiv*, abs/2310.15916, 2023. URL https://api.semanticscholar.org/CorpusID:264439386.

[60] G. E. Hinton and I. Sutskever. Training recurrent neural networks. 2013.

[61] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9:1735–1780, 1997.

[62] A. Hojel, Y. Bai, T. Darrell, A. Globerson, and A. Bar. Finding visual task vectors. *ArXiv*, abs/2404.05729, 2024. URL https://api.semanticscholar.org/CorpusID:269005382.

[63] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, W. Chen, et al. Lora: Low-rank adaptation of large language models. *ICLR*, 1(2):3, 2022.

[64] E. S. Hu, K. Ahn, Q. Liu, H. Xu, M. Tomar, A. Langford, D. Jayaraman, A. Lamb, and J. Langford. The belief state transformer. 2024. URL https://api.semanticscholar.org/CorpusID:273707334.

[65] S.-C. Huang, P.-Z. Li, Y.-C. Hsu, K.-M. Chen, Y. T. Lin, S.-K. Hsiao, R. T.-H. Tsai, and H. yi Lee. Chat vector: A simple approach to equip llms with

instruction following and model alignment in new languages. In *Annual Meeting of the Association for Computational Linguistics*, 2023. URL https://api.semanticscholar.org/CorpusID:268253329.

[66] H. Hwang, B. Jeon, S. Kim, J. Kim, H. Chang, S. Yang, S. Won, D. Lee, Y. Ahn, and M. Seo. Let's predict sentence by sentence, 2025. URL http://arxiv.org/abs/2505.22202. version: 1.

[67] G. Ilharco, M. T. Ribeiro, M. Wortsman, S. Gururangan, L. Schmidt, H. Hajishirzi, and A. Farhadi. Editing models with task arithmetic. *ArXiv*, abs/2212.04089, 2022. URL https://api.semanticscholar.org/CorpusID:254408495.

[68] H. Jaeger. A tutorial on training recurrent neural networks , covering bppt , rtrl , ekf and the " echo state network " approach - semantic scholar. 2005.

[69] A. Jaegle, F. Gimeno, A. Brock, A. Zisserman, O. Vinyals, and J. Carreira. Perceiver: General perception with iterative attention. In *International Conference on Machine Learning*, 2021.

[70] A. Jain and B. Rappazzo. Learning to reason with mixture of tokens. *arXiv preprint arXiv:2509.21482*, 2025.

[71] P. Kalyan, P. Rao, P. Jyothi, and P. Bhattacharyya. Emotion arithmetic: Emotional speech synthesis via weight space interpolation. *Interspeech 2024*, 2024. URL https://api.semanticscholar.org/CorpusID:271741273.

[72] J.-H. Kang, J. Lee, M.-H. Lee, and J.-H. Chang. Whisper multilingual downstream task tuning using task vectors. *Interspeech 2024*, 2024. URL https://api.semanticscholar.org/CorpusID:272332568.

[73] J. Kim, H. J. Kim, H. Cho, H. Jo, S.-W. Lee, S. goo Lee, K. M. Yoo, and T. Kim. Ground-truth labels matter: A deeper look into input-label demonstrations.

ArXiv, abs/2205.12685, 2022. URL https://api.semanticscholar.org/CorpusID:249062718.

[74] S. Kobayashi, S. Schug, Y. Akram, F. Redhardt, J. von Oswald, R. Pascanu, G. Lajoie, and J. Sacramento. When can transformers compositionally generalize in-context? *ArXiv*, abs/2407.12275, 2024. URL https://api.semanticscholar.org/CorpusID:271244618.

[75] J. Kossen, Y. Gal, and T. Rainforth. In-context learning learns label relationships but is not conventional learning. 2023. URL https://api.semanticscholar.org/CorpusID:260125327.

[76] V. Lad, W. Gurnee, and M. Tegmark. The remarkable robustness of llms: Stages of inference? *ArXiv*, abs/2406.19384, 2024. URL https://api.semanticscholar.org/CorpusID:270764625.

[77] Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, and R. Soricut. Albert: A lite bert for self-supervised learning of language representations. *ArXiv*, abs/1909.11942, 2019.

[78] J. Lee, A. Xie, A. Pacchiano, Y. Chandak, C. Finn, O. Nachum, and E. Brunskill. Supervised pretraining can learn in-context reinforcement learning. *ArXiv*, abs/2306.14892, 2023. URL https://api.semanticscholar.org/CorpusID:259262142.

[79] B. Lester, R. Al-Rfou, and N. Constant. The power of scale for parameter-efficient prompt tuning. In *Conference on Empirical Methods in Natural Language Processing*, 2021. URL https://api.semanticscholar.org/CorpusID:233296808.

[80] D. Li, Z. Liu, X. Hu, Z. Sun, B. Hu, and M. Zhang. In-context learning state vector with inner and momentum optimization. 2024. URL https://api.semanticscholar.org/CorpusID:269187789.

[81] X. L. Li and P. Liang. Prefix-tuning: Optimizing continuous prompts for generation. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 4582–4597, 2021.

[82] Y. Li, M. E. Ildiz, D. Papailiopoulos, and S. Oymak. Transformers as algorithms: Generalization and stability in in-context learning. 2023.

[83] Z. Li, Z. Xu, L. Han, Y. Gao, S. Wen, D. Liu, H. Wang, and D. N. Metaxas. Implicit in-context learning. *ArXiv*, abs/2405.14660, 2024. URL https://api.semanticscholar.org/CorpusID:269983389.

[84] Y.-H. Liao, S. Elflein, L. He, L. Leal-Taixé, Y. Choi, S. Fidler, and D. Acuna. Longperceptualthoughts: Distilling system-2 reasoning for system-1 perception. *arXiv preprint arXiv:2504.15362*, 2025.

[85] L. Lin, Y. Bai, and S. Mei. Transformers as decision makers: Provable in-context reinforcement learning via supervised pretraining. *ArXiv*, abs/2310.08566, 2023. URL https://api.semanticscholar.org/CorpusID:263909278.

[86] Z. Lin and K. Lee. Dual operating modes of in-context learning. *arXiv preprint arXiv:2402.18819*, 2024.

[87] D. Lioubashevski, T. M. Schlank, G. Stanovsky, and A. Goldstein. Looking beyond the top-1: Transformers determine top tokens in order. *ArXiv*, abs/2410.20210, 2024. URL https://api.semanticscholar.org/CorpusID:273655045.

[88] M. Littman and R. S. Sutton. Predictive representations of state. *Advances in neural information processing systems*, 14, 2001.

[89] S. Liu, H. Ye, L. Xing, and J. Y. Zou. In-context vectors: Making in context learning more effective and controllable through latent space steering.

ArXiv, abs/2311.06668, 2023. URL https://api.semanticscholar.org/CorpusID:265149781.

[90] X. Liu, Y. Zheng, Z. Du, M. Ding, Y. Qian, Z. Yang, and J. Tang. Gpt understands, too. *ArXiv*, abs/2103.10385, 2021. URL https://api.semanticscholar.org/CorpusID:232269696.

[91] Z. Liu, J. Wang, T. Dao, T. Zhou, B. Yuan, Z. Song, A. Shrivastava, C. Zhang, Y. Tian, C. Ré, and B. Chen. Deja vu: Contextual sparsity for efficient llms at inference time. In *International Conference on Machine Learning*, 2023. URL https://api.semanticscholar.org/CorpusID:260815690.

[92] Q. Long, Y. Wu, W. Wang, and S. J. Pan. Decomposing label space, format and discrimination: Rethinking how llms respond and solve tasks via in-context learning. *ArXiv*, abs/2404.07546, 2024. URL https://api.semanticscholar.org/CorpusID:269042687.

[93] G. Luo, T. Darrell, and A. Bar. Task vectors are cross-modal. 2024. URL https://api.semanticscholar.org/CorpusID:273661988.

[94] A. Lv, K. Zhang, Y. Chen, Y. Wang, L. Liu, J.-R. Wen, J. Xie, and R. Yan. Interpreting key mechanisms of factual recall in transformer-based language models. *ArXiv*, abs/2403.19521, 2024. URL https://api.semanticscholar.org/CorpusID:268732668.

[95] A. V. Mahankali, T. Hashimoto, and T. Ma. One step of gradient descent is provably the optimal in-context learner with one layer of linear self-attention. *ArXiv*, abs/2307.03576, 2023.

[96] K. Meng, D. Bau, A. Andonian, and Y. Belinkov. Locating and editing factual associations in gpt. In *Neural Information Processing Systems*, 2022. URL https://api.semanticscholar.org/CorpusID:255825985.

[97] J. Merullo, C. Eickhoff, and E. Pavlick. Language models implement simple word2vec-style vector arithmetic. *ArXiv*, abs/2305.16130, 2023. URL https://api.semanticscholar.org/CorpusID:258887799.

[98] J. Merullo, C. Eickhoff, and E. Pavlick. A mechanism for solving relational tasks in transformer language models. 2023.

[99] S. Min, M. Lewis, L. Zettlemoyer, and H. Hajishirzi. Metaicl: Learning to learn in context. *ArXiv*, abs/2110.15943, 2021. URL https://api.semanticscholar.org/CorpusID:240288835.

[100] S. Min, X. Lyu, A. Holtzman, M. Artetxe, M. Lewis, H. Hajishirzi, and L. Zettlemoyer. Rethinking the role of demonstrations: What makes in-context learning work? In *Conference on Empirical Methods in Natural Language Processing*, 2022.

[101] S. Mittal, E. Elmoznino, L. Gagnon, S. Bhardwaj, D. Sridhar, and G. Lajoie. Does learning the right latent variables necessarily improve in-context learning? *ArXiv*, abs/2405.19162, 2024. URL https://api.semanticscholar.org/CorpusID:270095225.

[102] J. Mu, X. Li, and N. Goodman. Learning to compress prompts with gist tokens. *Advances in Neural Information Processing Systems*, 36:19327–19352, 2023.

[103] S. Muller, N. Hollmann, S. P. Arango, J. Grabocka, and F. Hutter. Transformers can do bayesian inference. *ArXiv*, abs/2112.10510, 2021.

[104] T. Ni, B. Eysenbach, E. Seyedsalehi, M. Ma, C. Gehring, A. Mahajan, and P.-L. Bacon. Bridging state and history representations: Understanding self-predictive rl. *ArXiv*, abs/2401.08898, 2024. URL https://api.semanticscholar.org/CorpusID:267027743.

[105] nostalgebraist. interpreting gpt: the logit lens., 2020. URL https://www.lesswrong.com/posts/AcKRB8wDpdaN6v6ru/interpreting-gpt-the-logit-lens.

[106] G. Ongie, A. Jalal, C. A. Metzler, R. Baraniuk, A. G. Dimakis, and R. M. Willett. Deep learning techniques for inverse problems in imaging. *IEEE Journal on Selected Areas in Information Theory*, 1:39–56, 2020.

[107] J. Pan, T. Gao, H. Chen, and D. Chen. What in-context learning "learns" in-context: Disentangling task recognition and task learning. In *Annual Meeting of the Association for Computational Linguistics*, 2023. URL https://api.semanticscholar.org/CorpusID:258740972.

[108] C. F. Park, A. Lee, E. S. Lubana, Y. Yang, M. Okawa, K. Nishi, M. Wattenberg, and H. Tanaka. Iclr: In-context learning of representations. 2024. URL https://api.semanticscholar.org/CorpusID:275212269.

[109] K. Park, Y. J. Choe, and V. Veitch. The linear representation hypothesis and the geometry of large language models. *ArXiv*, abs/2311.03658, 2023. URL https://api.semanticscholar.org/CorpusID:265042984.

[110] Y. Peng, C. Hao, X. Yang, J. Peng, X. Hu, and X. Geng. Learnable in-context vector for visual question answering. *ArXiv*, abs/2406.13185, 2024. URL https://api.semanticscholar.org/CorpusID:270619372.

[111] J. Pfau, W. Merrill, and S. R. Bowman. Let's think dot by dot: Hidden computation in transformer language models, 2024. URL http://arxiv.org/abs/2404.15758.

[112] M. Pham, K. O. Marshall, C. Hegde, and N. Cohen. Robust concept erasure using task vectors. *ArXiv*, abs/2404.03631, 2024. URL https://api.semanticscholar.org/CorpusID:268889354.

[113] J. Phang. Investigating the effectiveness of hypertuning via gisting. *ArXiv*, abs/2402.16817, 2024. URL https://api.semanticscholar.org/CorpusID:268032098.

[114] S. T. Piantadosi, D. C. Muller, J. S. Rule, K. Kaushik, M. I. Gorenstein, E. R. Leib, and E. Sanford. Why concepts are (probably) vectors. *Trends in Cognitive Sciences*, 28:844–856, 2024. URL https://api.semanticscholar.org/CorpusID:271721102.

[115] Qwen, :, A. Yang, B. Yang, B. Zhang, B. Hui, B. Zheng, B. Yu, C. Li, D. Liu, F. Huang, H. Wei, H. Lin, J. Yang, J. Tu, J. Zhang, J. Yang, J. Yang, J. Zhou, J. Lin, K. Dang, K. Lu, K. Bao, K. Yang, L. Yu, M. Li, M. Xue, P. Zhang, Q. Zhu, R. Men, R. Lin, T. Li, T. Tang, T. Xia, X. Ren, X. Ren, Y. Fan, Y. Su, Y. Zhang, Y. Wan, Y. Liu, Z. Cui, Z. Zhang, and Z. Qiu. Qwen2.5 technical report, 2025. URL https://arxiv.org/abs/2412.15115.

[116] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, I. Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

[117] A. Raventos, M. Paul, F. Chen, and S. Ganguli. The effects of pretraining task diversity on in-context learning of ridge regression. In *ICLR 2023 Workshop on Mathematical and Empirical Understanding of Foundation Models*, 2023.

[118] S. Ren, Q. Jia, and K. Q. Zhu. Context compression for auto-regressive transformers with sentinel tokens. In *Conference on Empirical Methods in Natural Language Processing*, 2023. URL https://api.semanticscholar.org/CorpusID:263909553.

[119] B. Saglam, Z. Yang, D. Kalogerias, and A. Karbasi. Learning task representations from in-context learning. In *ICML 2024 Workshop on In-Context Learning*.

[120] C. Sanford, B. Fatemi, E. Hall, A. Tsitsulin, M. Kazemi, J. Halcrow, B. Perozzi, and V. Mirrokni. Understanding Transformer Reasoning Capabilities

via Graph Algorithms, May 2024. URL http://arxiv.org/abs/2405.18512. arXiv:2405.18512 [cs].

[121] A. Saparov and H. He. Language models are greedy reasoners: A systematic formal analysis of chain-of-thought. *arXiv preprint arXiv:2210.01240*, 2022.

[122] N. Saunshi, N. Dikkala, Z. Li, S. Kumar, and S. J. Reddi. Reasoning with latent thoughts: On the power of looped transformers. *arXiv preprint arXiv:2502.17416*, 2025.

[123] A. Schwarzschild, E. Borgnia, A. Gupta, F. Huang, U. Vishkin, M. Goldblum, and T. Goldstein. Can you learn an algorithm? generalizing from easy to hard problems with recurrent networks. In *Advances in Neural Information Processing Systems*, 2021.

[124] Z. Shao, P. Wang, Q. Zhu, R. Xu, J. Song, X. Bi, H. Zhang, M. Zhang, Y. Li, Y. Wu, et al. Deepseekmath: Pushing the limits of mathematical reasoning in open language models. *arXiv preprint arXiv:2402.03300*, 2024.

[125] P. Sharma, J. T. Ash, and D. Misra. The truth is in there: Improving reasoning in language models with layer-selective rank reduction. *ArXiv*, abs/2312.13558, 2023. URL https://api.semanticscholar.org/CorpusID:266435969.

[126] S. Z. Shen, R. Shao, C. Wang, S. Yang, V.-P. Berges, G. Ghosh, P. W. Koh, L. Zettlemoyer, Y. Kim, J. E. Weston, D. Sontag, and W.-t. Yih. Hybrid-cot: Interleaving latent and text chain-of-thought for efficient reasoning. In *NeurIPS 2025 ER Workshop*, 2025. Spotlight presentation.

[127] Z. Shen, H. Yan, L. Zhang, Z. Hu, Y. Du, and Y. He. Codi: Compressing chain-of-thought into continuous space via self-distillation. *arXiv preprint arXiv:2502.21074*, 2025.

[128] S. Sia, D. Mueller, and K. Duh. Where does in-context translation happen in large language models. *ArXiv*, abs/2403.04510, 2024. URL https://api.semanticscholar.org/CorpusID:268264275.

[129] A. K. Singh, T. Moskovitz, F. Hill, S. C. Y. Chan, and A. M. Saxe. What needs to go right for an induction head? a mechanistic study of in-context learning circuits and their formation. *ArXiv*, abs/2404.07129, 2024. URL https://api.semanticscholar.org/CorpusID:269033179.

[130] D. Su, S. Sukhbaatar, M. Rabbat, Y. Tian, and Q. Zheng. Dualformer: Controllable fast and slow thinking by learning with randomized reasoning traces. *arXiv preprint arXiv:2410.09918*, 2024.

[131] D. Su, H. Zhu, Y. Xu, J. Jiao, Y. Tian, and Q. Zheng. Token assorted: Mixing latent and text tokens for improved language model reasoning. *arXiv preprint arXiv:2502.03275*, 2025.

[132] Y. Sun, Y. Chen, Y. Li, and B. Ding. Enhancing latent computation in transformers with latent tokens, 2025. URL http://arxiv.org/abs/2505.12629.

[133] S. K. Swaminathan, A. Dedieu, R. V. Raju, M. Shanahan, M. Lázaro-Gredilla, and D. George. Schema-learning and rebinding as mechanisms of in-context learning and emergence. *ArXiv*, abs/2307.01201, 2023. URL https://api.semanticscholar.org/CorpusID:259342490.

[134] J. Tack, J. Lanchantin, J. Yu, A. Cohen, I. Kulikov, J. Lan, S. Hao, Y. Tian, J. Weston, and X. Li. LLM Pretraining with Continuous Concepts, Feb. 2025. URL http://arxiv.org/abs/2502.08524. arXiv:2502.08524 [cs].

[135] W. Tan, J. Li, J. Ju, Z. Luo, J. Luan, and R. Song. Think silently, think fast: Dynamic latent compression of llm reasoning chains. *arXiv preprint arXiv:2505.16552*, 2025.

[136] Z. Tao, I. Mason, S. Kulkarni, and X. Boix. Task arithmetic through the lens of one-shot federated learning. 2024. URL https://api.semanticscholar.org/CorpusID:274306177.

[137] G. Team, A. Kamath, J. Ferret, S. Pathak, N. Vieillard, R. Merhej, S. Perrin, T. Matejovicova, A. Ramé, M. Rivière, et al. Gemma 3 technical report. *arXiv preprint arXiv:2503.19786*, 2025.

[138] R. Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the royal statistical society series b-methodological*, 58:267–288, 1996.

[139] E. Todd, M. Li, A. S. Sharma, A. Mueller, B. C. Wallace, and D. Bau. Function vectors in large language models. *ArXiv*, abs/2310.15213, 2023. URL https://api.semanticscholar.org/CorpusID:264439657.

[140] J. Vanschoren, J. N. van Rijn, B. Bischl, and L. Torgo. Openml: networked science in machine learning. *SIGKDD Explorations*, 15(2):49–60, 2013. doi: 10.1145/2641190.2641198. URL http://doi.acm.org/10.1145/2641190.264119.

[141] J. von Oswald, E. Niklasson, E. Randazzo, J. Sacramento, A. Mordvintsev, A. Zhmoginov, and M. Vladymyrov. Transformers learn in-context by gradient descent. *ArXiv*, abs/2212.07677, 2022.

[142] J. von Oswald, E. Niklasson, M. Schlegel, S. Kobayashi, N. Zucchet, N. Scherrer, N. Miller, M. Sandler, B. A. y Arcas, M. Vladymyrov, R. Pascanu, and J. Sacramento. Uncovering mesa-optimization algorithms in transformers. *ArXiv*, abs/2309.05858, 2023. URL https://api.semanticscholar.org/CorpusID:261696852.

[143] K. Wang, A. Variengien, A. Conmy, B. Shlegeris, and J. Steinhardt. Interpretability in the wild: a circuit for indirect object identification in gpt-2 small. *ArXiv*, abs/2211.00593, 2022. URL https://api.semanticscholar.org/CorpusID:253244237.

[144] L. Wang, L. Li, D. Dai, D. Chen, H. Zhou, F. Meng, J. Zhou, and X. Sun. Label words are anchors: An information flow perspective for understanding in-context learning. In *Conference on Empirical Methods in Natural Language Processing*, 2023. URL https://api.semanticscholar.org/CorpusID:258841117.

[145] X. Wang, S. Wang, Y. Zhu, and B. Liu. System-1.5 reasoning: Traversal in language and latent spaces with dynamic shortcuts. *arXiv preprint arXiv:2505.18962*, 2025.

[146] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

[147] C. Wendler, V. Veselovsky, G. Monea, and R. West. Do llamas work in english? on the latent language of multilingual transformers. *ArXiv*, abs/2402.10588, 2024. URL https://api.semanticscholar.org/CorpusID:267740723.

[148] N. Wies, Y. Levine, and A. Shashua. The learnability of in-context learning. In A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, editors, *Advances in Neural Information Processing Systems*, volume 36, pages 36637–36651. Curran Associates, Inc., 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/73950f0eb4ac0925dc71ba2406893320-Paper-Conference.pdf.

[149] E. Winston and J. Z. Kolter. Monotone operator equilibrium networks. *ArXiv*, abs/2006.08591, 2020.

[150] C. Wu, J. Lu, Z. Ren, G. Hu, Z. Wu, D. Dai, and H. Wu. Llms are single-threaded reasoners: Demystifying the working mechanism of soft thinking. *arXiv preprint arXiv:2508.03440*, 2025.

[151] S. Wu, Y. Wang, and Q. Yao. Why in-context learning models are good few-shot learners? In *International Conference on Learning Representations (ICLR)*, 2025. URL https://openreview.net/forum?id=iLUcsecZJp.

[152] H. Xia, C. T. Leong, W. Wang, Y. Li, and W. Li. Tokenskip: Controllable chain-of-thought compression in llms. *arXiv preprint arXiv:2502.12067*, 2025.

[153] S. M. Xie, A. Raghunathan, P. Liang, and T. Ma. An explanation of in-context learning as implicit bayesian inference. *ArXiv*, abs/2111.02080, 2021.

[154] Z. Xiong, Z. Cai, J. Cooper, A. Ge, V. Papageorgiou, Z. Sifakis, A. Giannou, Z. Lin, L. Yang, S. Agarwal, et al. Everything everywhere all at once: Llms can in-context learn multiple tasks in superposition. *arXiv preprint arXiv:2410.05603*, 2024.

[155] Z. Xiong, S. Garg, V. Shrivastava, H. Zhao, A. Kyrillidis, and D. Papailiopoulos. Superposition reasoning model. In *NeurIPS 2025 Workshop on Efficient Reasoning*, 2025. URL https://openreview.net/forum?id=TZxPjnYDBI.

[156] Z. Xu, Z. Liu, B. Chen, Y. Tang, J. Wang, K. Zhou, X. Hu, and A. Shrivastava. Compress, then prompt: Improving accuracy-efficiency trade-off of llm inference with transferable prompt. *ArXiv*, abs/2305.11186, 2023. URL https://api.semanticscholar.org/CorpusID:258823240.

[157] S. Yadlowsky, L. Doshi, and N. Tripuraneni. Pretraining data mixtures enable narrow model selection capabilities in transformer models. *ArXiv*, abs/2311.00871, 2023. URL https://api.semanticscholar.org/CorpusID:264935329.

[158] S. Yadlowsky, L. Doshi, and N. Tripuraneni. Can transformer models generalize via in-context learning beyond pretraining data? In *NeurIPS 2023 Workshop on Distribution Shifts: New Frontiers with Foundation Models*, 2023.

[159] Y. Yan, Y. Shen, Y. Liu, J. Jiang, M. Zhang, J. Shao, and Y. Zhuang. Inftythink: Breaking the length limits of long-context reasoning in large language models. *arXiv preprint arXiv:2503.06692*, 2025.

[160] C. Yang, N. Srebro, D. McAllester, and Z. Li. PENCIL: Long thoughts with short memory, 2025. URL http://arxiv.org/abs/2503.14337.

[161] L. Yang, K. Lee, R. D. Nowak, and D. Papailiopoulos. Looped transformers are better at learning learning algorithms. In *The Twelfth International Conference on Learning Representations*, 2024.

[162] L. Yang, Z. Lin, K. Lee, D. Papailiopoulos, and R. Nowak. Task vectors in in-context learning: Emergence, formation, and benefit. *arXiv preprint arXiv:2501.09240*, 2025.

[163] Q. Ye, B. Y. Lin, and X. Ren. CrossFit: A few-shot learning challenge for cross-task generalization in NLP. In M.-F. Moens, X. Huang, L. Specia, and S. W.-t. Yih, editors, *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 7163–7189, Online and Punta Cana, Dominican Republic, Nov. 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.572. URL https://aclanthology.org/2021.emnlp-main.572/.

[164] T. Ye, Z. Xu, Y. Li, and Z. Allen-Zhu. Physics of Language Models: Part 2.1, Grade-School Math and the Hidden Reasoning Process. *ArXiv e-prints*, abs/2407.20311, July 2024. Full version available at http://arxiv.org/abs/2407.20311.

[165] P. Yu, J. Xu, J. Weston, and I. Kulikov. Distilling system 2 into system 1. *arXiv preprint arXiv:2407.06023*, 2024.

[166] Q. Yu, J. Merullo, and E. Pavlick. Characterizing mechanisms for factual recall in language models. *ArXiv*, abs/2310.15910, 2023. URL https://api.semanticscholar.org/CorpusID:264439114.

[167] Z. Yue, B. Jin, H. Zeng, H. Zhuang, Z. Qin, J. Yoon, L. Shang, J. Han, and D. Wang. Hybrid latent reasoning via reinforcement learning. URL http://arxiv.org/abs/2505.18454.

[168] J. Yun and J. Choo. Scaling up personalized aesthetic assessment via task vector customization. *ArXiv*, abs/2407.07176, 2024. URL https://api.semanticscholar.org/CorpusID:271088904.

[169] J. Zhang, Y. Zhu, M. Sun, Y. Luo, S. Qiao, L. Du, D. Zheng, H. Chen, and N. Zhang. Lightthinker: Thinking step-by-step compression. *arXiv preprint arXiv:2502.15589*, 2025.

[170] R. Zhang, S. Frei, and P. L. Bartlett. Trained transformers learn linear models in-context. *ArXiv*, abs/2306.09927, 2023.

[171] R. Zhang, J. Wu, and P. L. Bartlett. In-context learning of a linear transformer block: Benefits of the mlp component and one-step gd initialization. *arXiv preprint arXiv:2402.14951*, 2024.

[172] Y. Zhang, F. Zhang, Z. Yang, and Z. Wang. What and how does in-context learning learn? bayesian model averaging, parameterization, and generalization. *ArXiv*, abs/2305.19420, 2023. URL https://api.semanticscholar.org/CorpusID:258987402.

[173] Z. Zhang, X. He, W. Yan, A. Shen, C. Zhao, S. Wang, Y. Shen, and X. E. Wang. Soft thinking: Unlocking the reasoning potential of LLMs in continuous concept space, 2025. URL http://arxiv.org/abs/2505.15778.

[174] H. Zhao, H. Zhao, B. Shen, A. Payani, F. Yang, and M. Du. Beyond single concept vector: Modeling concept subspace in llms with gaussian distribution. 2024. URL https://api.semanticscholar.org/CorpusID:273022717.

[175] H. Zhu, S. Hao, Z. Hu, J. Jiao, S. Russell, and Y. Tian. Reasoning by superposition: A theoretical perspective on chain of continuous thought. *arXiv preprint arXiv:2505.12514*, 2025.

[176] R. Zhu, H. Zhang, T. Shi, C. Wang, T. Zhou, and Z. Qin. The 4th dimension for scaling model size. *arXiv preprint arXiv:2506.18233*, 2025.

[177] R.-J. Zhu, Z. Wang, K. Hua, T. Zhang, Z. Li, H. Que, B. Wei, Z. Wen, F. Yin, H. Xing, et al. Scaling latent reasoning via looped language models. *arXiv preprint arXiv:2510.25741*, 2025.

[178] Y. Zhuang, L. Liu, C. Singh, J. Shang, and J. Gao. Text generation beyond discrete token sampling. *arXiv preprint arXiv:2505.14827*, 2025.