Compressed Chain of Thought: Efficient Reasoning through Dense Representations

Jeffrey Cheng [1] **Benjamin Van Durme** [1]

Abstract

Chain-of-thought (CoT) decoding enables language models to improve reasoning performance at the cost of high generation latency in decoding. Recent proposals have explored variants of *contemplation tokens*, a term we introduce that refers to special tokens used during inference to allow for extra computation. Prior work has considered fixed-length sequences drawn from a *discrete* set of embeddings as contemplation tokens. Here we propose Compressed Chain-of-Thought (CCoT), a framework to generate *contentful* and *continuous* contemplation tokens of variable sequence length. The generated contemplation tokens are compressed representations of explicit reasoning chains, and our method can be applied to off-the-shelf decoder language models. Through experiments, we illustrate how CCoT enables additional reasoning over dense contentful representations to achieve corresponding improvements in accuracy. Moreover, the reasoning improvements can be adaptively modified on demand by controlling the number of contemplation tokens generated.

1. Introduction

Chain-of-Thought (CoT) refers to the Large Language Model (LLM) technique in which the model simulates the process of thinking out loud by decomposing a complex question into parts and sequentially reasoning through each step. This behavior can be induced by finetuning on a dataset or human feedback (Liu et al., 2023; Puerto et al., 2024), demonstrating through ICL (Wei et al., 2023), or by providing tuned model instructions (Kojima et al., 2023). While CoT improves the reasoning capabilities of LLMs on a variety of tasks, the improvements come at the cost of a high generation latency. For instance, GPT-4o takes 21.37 seconds to generate a response to the question shown in Figure 1 with CoT prompting, whereas it can answer the same question without CoT prompting in 2.81 seconds, achieving the same answer with an almost 10x speedup.

Past work has utilized what we term *contemplation tokens* as an alternative to explicit CoT reasoning traces (Pfau et al., 2024; Goyal et al., 2024). These are additional tokens used to introduce online memory, allowing for additional computations during inference. Instead of generating a reasoning chain entirely of explicit language tokens, the model conditions on a shorter sequence of contemplation tokens (Section 2). Contemplation tokens can either be *contentful*, grounded in semantically meaningful text, or *noncontentful*. There are many lines of prior work involving *noncontentful* contemplation tokens drawn from a set of *discrete* tokens; this paper introduces *contentful* contemplation tokens that represent reasoning chains performed in *continuous* space.

Our framework, called Compressed Chain of Thought (CCoT), generates contemplation tokens which are compressed representations of language-based reasoning chains. These contemplation tokens are trained through teacher forcing with respect to the gold hidden states corresponding to full reasoning traces. Our framework can be adapted to pretrained LLMs through LoRA finetuning. Moreover, the variable compression ratio during training allows for need-based adjustments to the performance-efficiency tradeoff by controlling the number of tokens generated during inference.

The contributions of this paper are as follows:

1. We finetune pretrained decoder-only LLMs with our new CCOT framework and empirically evaluate their performance and throughput on GSM8K;
2. We establish our framework in context of related work in filler tokens and CoT distillation in terms of performance and efficiency;
3. We extend theoretical results and demonstrate the computational capacity of CCOT contemplations tokens.

2. Related Work

Distillation of Knowledge Chains There has been work in distilling the computations done explicitly when decoding the reasoning chains into computation of the hidden states

[1]Department of Computer Science, Johns Hopkins University, Baltimore, US. Correspondence to: Jeffrey Cheng <jcheng71@jh.edu>.

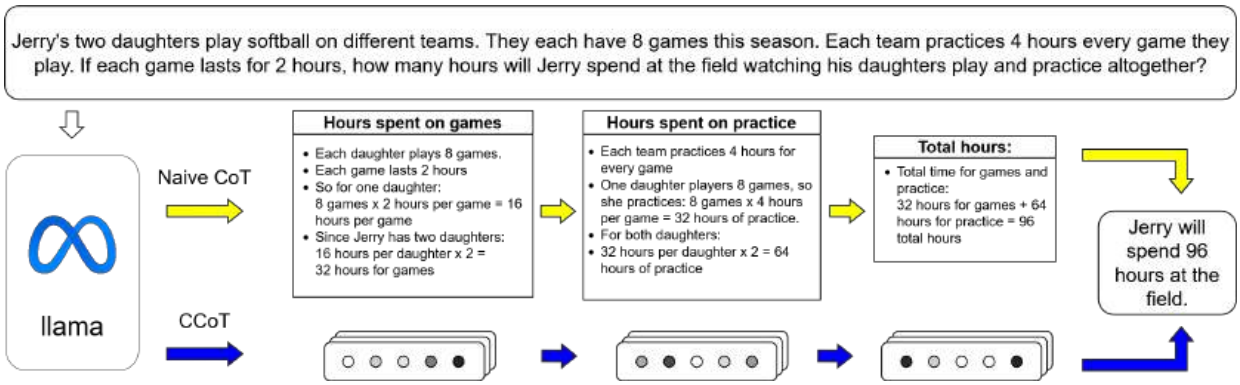


Figure 1. Two approaches to step by step reasoning. Chain of Thought (CoT) prompting reasons via discrete language tokens, leading to long sequences that incur significant generation costs. In contrast Compressed Chain of Thought (CCOT) elicits reasoning with a short sequence of continuous embeddings, allowing for much greater throughput.

of the answer (Deng et al., 2023; 2024). Contemporaneous work distills reasoning paths into continuous latent tokens (Hao et al., 2024). Our method differs in that the contemplation tokens we generate are grounded in text rather than only used as a signal to decode from. This is a critical distinction: our grounding offers the future potential for decoding the reasoning chain from the compressed representations, allowing for post-hoc human inspection of the LLM's reasoning. Moreover, our method successfully adapts a much larger model (7B compared to 1.5B) using a fraction of data (≈ 9000 instances in GSM8K compared to ≈ 400000 instances in an unreleased augmented GSM8K). This suggests that our method can scale better to larger models and is more data efficient.

Filler (Pause) Tokens Many previous methods have considered decoding *contemplation tokens* to provide an LLM with more compute during inference time. These tokens have gone by many names, such as pause tokens (Goyal et al., 2024), memory tokens (Burtsev et al., 2021), filler tokens (Pfau et al., 2024), and thinking tokens (Herel & Mikolov, 2024). These works mainly focus on *noncontentful* contemplation tokens, whose main advantage is their ability to be decoded in parallel, providing the model with a greater computational width without the need to autoregressively decode.

They have been shown to increase the theoretical computational ability of Transformer LLMs (Pfau et al., 2024), but cannot simply be naively applied to induce reasoning gains (Lanham et al., 2023). However, through careful pretraining and finetuning, pause tokens have been shown to improve reasoning in both RNNs (Herel & Mikolov, 2024) and Transformers (Goyal et al., 2024). In contrast, the contemplation tokens generated by CCOT are contentful as they are compressed representations of reasoning chains. Moreover, they are decoded autoregressively resulting in a greater computational depth as well as width.

Contextual Compression Transformer LLMs are the de facto standard architecture for modern NLP applications. However, due to the quadratic complexity of its self-attention mechanism, these LLMs are inefficient in tasks with long contexts. Many techniques have been proposed to alleviate this issue, including memory slots (Ge et al., 2024), dynamic compression into nuggets (Qin et al., 2024), and low level cache encoding (Liu et al., 2024). While most techniques rely on access to the intermediate hidden states of LLMs, there has also been work done in the context of API-only LLMs (Jiang et al., 2023). Overall, most of the work in contextual compression deals with efficient compression of known context in order to improve generation latency. The compressed context can then be used in downstream tasks such as retrieval augmented generation or summarization.

The area of context compression is orthogonal to *comptemplation tokens*. The memory slots of (Ge et al., 2024) and the nuggets from (Qin et al., 2024) encode contentful representations of *known* context, but they are only attended to and never generated during inference. While our work focuses on contentful representations of text, there are two crucial differences: our compressed representations are autoregressively *decoded* during inference and they encode content that is a priori *unknown*.

Chain of Thought Chain-of-thought (Wei et al., 2023) was introduced as a prompting method leveraging in-context learning (ICL) using hand crafted demonstrations. Kojima et al. (2023) showed similar behavior could be elicited in a zero-shot context by instructing a model to "think step-by-step." There have been a variety of innovations to CoT, improving on its efficiency and performance.

In terms of efficiency, novel techniques include getting an LLM to generate steps in parallel from a generated tem-

Method	Contentful	Format	Inference	Additional Notes
Chain of Thought (Wei et al., 2023)	Yes	Discrete	Variable-length; Autoregressively	Best performing method across reasoning tasks; requires no finetuning; inefficient due to unconstrained sequence length.
Filler Tokens (Pfau et al., 2024)	No	Discrete	Fixed-length; In parallel	Explicit example of problems only solvable with contemplation tokens.
Pause Tokens (Goyal et al., 2024)	No	Discrete	Fixed-length; In parallel	Best gains seen when contemplation tokens are added during pretraining stage.
COCONUT (Hao et al., 2024)	Yes	Continuous	Fixed-length; Autoregressively	Trains contemplation tokens by inserting them after removing reasoning steps.
CCOT (**Ours**)	Yes	Continuous	Variable-length; Autoregressively	Trains contemplation tokens to approximate compressed reasoning chains.

Table 1. A comparison of different methods to generate *contemplation tokens* in order to introduce extra computation into models during inference. We characterize several aspects of the tokens: (1) contentful, the tokens are either intrinsically contentful or approximate/are distilled from contentful text; (2) format, whether the tokens are drawn from a discrete set of embeddings or are drawn from continuous space; and (3) inference, how the tokens are generated during inference. Any additional notes for each method are included as well.

plate (Ning et al., 2024) and generating reasoning chains in parallel using Jacobi decoding (Kou et al., 2024; Zhang et al., 2024). In terms of performance, techniques include generating multiple reasoning paths (Yao et al., 2023), and finetuning on human feedback on generated chains (Liu et al., 2023; Puerto et al., 2024). Our method differs from prior work in improving the efficiency of CoT as it is not prompt-based and does not rely on Jacobi decoding.

3. Contemplation Tokens

3.1. Preliminaries and Notation

We first give a brief overview of a causal decoder-only language model, equipped with standard Transformer blocks (Vaswani et al., 2023). Let V be the vocabulary and $w_{1:n}$ be an input sequence, $w_i \in V$. Let d be the hidden dimension, L be the number of layers, and θ be the parameters of the model. The sequence is first passed through an embedding layer, resulting in a vector $w^0_{1:n}$ where each $w^0_i \in \mathbb{R}^d$. The entire vector $w^0_{1:n} \in \mathbb{R}^{n\times d}$ is then passed through a series of Transformer blocks, $T^i : \mathbb{R}^{n\times d} \to \mathbb{R}^{n\times d}$. We denote the output of each T^i as the *hidden states*. The output of the final Transformer block, $w^L_{1:n} \in \mathbb{R}^{n\times d}$, is then passed through the language model head to generate a distribution $p_{1:n}$, $p_i \in \mathbb{R}^{|V|}$, from which the next token is sampled.

$$
\begin{aligned}
w^0_{1:n} &= \textsc{embed}_\theta(w_{1:n}) && \triangleright \textit{embedding layer} \\
w^\ell_{1:n} &= \textsc{attn}^{\ell-1}_\theta(w^{\ell-1}_{1:n}) && \triangleright \textit{transformer blocks} \\
p_{1:n} &= \textsc{head}_\theta(w^L_{1:n}) && \triangleright \textit{pass through lm head} \\
p(w_{n+1} \mid w_{1:n}) &\sim p_n && \triangleright \textit{sample next token}
\end{aligned}
$$

Notation-wise, any lowercase letter will refer to a *token*, lying in V. Any lowercase letter with superscripts will refer to the hidden state after passing through the corresponding layer, lying in $\mathbb{R}^d$. Any subscripts refers to a sequence. We will often omit superscripts and instead refer to embeddings with bars ($\bar{}$) and the entire hidden state with hats ($\hat{}$). Under this notation, we instead have $\textsc{embed}(w_{1:n}) = \bar{w}_{1:n}$, and with slight abuse of notation, $\textsc{attn}(\bar{w}_{1:n}) = \hat{w}_{1:n}$.

There are also instances where hidden states of an input are computed under two different sets of weights. Suppose we have two sequences of embeddings $\bar{w}$, $\bar{x}$, and we want to compute the hidden states $\hat{w}$ under weights θ and compute the hidden states of $\hat{x}$ under ψ, but crucially conditioned on $\hat{w}$. In this case, we will write $\textsc{attn}_{\theta,\psi}([\bar{w};\bar{x}]) = [\hat{w};\hat{x}]$ where semicolons indicate vector concatenation.

3.2. Motivation

In question-answer settings, the input $w_{1:n}$ is a query, and the answer $w_{n+1:n+o} = a_{1:o}$ is generated autoregressively as described above. However as seen in the above description of forward passes through Transformer models, the amount of computations for each query is directly proportional to the query length n. As such, we can introduce more computations to the model by attending to an a set of *contemplation tokens*, defined to be any additional tokens generated during inference used to introduce addition memory allowing for additional computations during inference. Rather than solely attending to a query $q = w_{1:n}$, we first can generate a set of contemplation tokens $t = t_{1:m}$ and attend to $[q;t]$ in order to decode a better answer. We emphasize that contemplation tokens are not a novel idea, but a term introduced to unify the many names given to this

concept (Section 2).

We define the contemplation tokens t to be *contentful* if either the tokens themselves are semantically contentful or the hidden states corresponding to the contemplation tokens are derived from semantically contentful tokens. We define contemplation tokens that do not fulfill either of these conditions to be *noncontentful*. An example of contentful contemplation tokens are the reasoning chains in chain of thought (Wei et al., 2023); they describe the model's reasoning, fulfilling the first condition of being semantically meaningful. On the other hand, an example of noncontentful contemplation tokens are filler tokens (Pfau et al., 2024), as they are simply period characters and their hidden states are trained without any signal from semantically contentful hidden states.

Chain of thought turns out to be the only prior method involving contentful contemplation tokens. The performance gains from utilizing chain of thought are clear; however these benefits are offset by the high generation latency. Suppose the input query consists of n tokens and its corresponding reasoning chain consists of m tokens. As each of the tokens in the reasoning chain need to be autoregressively decoded, the generation of the reasoning chain incurs the cost of m extra passes through the model. Moreover when decoding the answer, the model has to attend to the additional m tokens, resulting in $O(m^2)$ more computations when passing through each attention module. As reasoning chains are often many times longer than the query, the amount of extra computations increases dramatically.[1]

3.3. Compressing Reasoning Chains

Prior work showed that *noncontentful* contemplation tokens only improved reasoning when the task was computationally bottlenecked (Pfau et al., 2024) or when the tokens were introduced during pretraining (Goyal et al., 2024). We instead aim to utilize *contentful* contemplation tokens as we believe they would be more applicable to a wider set of tasks. To generate contentful contemplation tokens, we take inspiration from an empirical observation of CoT decoding.

Suppose we have an input query $w_{1:n}$ and its corresponding reasoning chain $t_{1:m}$. We compute the hidden states of concatenated input as $x = [\hat{w}_{1:n}; \hat{t}_{1:m}]$. Decoding an answer conditioned on the hidden states x is equivalent to prompting a language model with the query and chain of thought. Consider taking a subset of $\hat{t}_{1:m}$ along the sequence length axis, denoted as $z_{1:k}$ for some $k << m$. Specifically, for each $1 \leq i \leq k$, there exists some $1 \leq j \leq m$ such that $z_i = \hat{t}_j$ at each layer. We observe that training an adapter to decode conditioning on this shortened $[x_{1:n}; z_{1:k}]$ results in lossless performance on downstream tasks.

Given a query q, the naive method utilizing this observation would be to autoregressively generate the reasoning chain t, select some learned subset of the encoded hidden states z, and train an adapter to decode from the query and subset of hidden states. While this method results in a shorter input sequence when generating the answer and thus reduces the attention computations when decoding the answer, it would still incur the linear cost in generating the reasoning chain. We instead propose learning a module to generate the compressed representations z directly. We denote this module as CCOT, short for **c**ompressed **c**hain **o**f **t**hought, as the contemplation tokens it generates are compressed representations of reasoning chains instead of the full chain.

4. Approach

Assume we have a pretrained causal decoder-only language model LM, parameterized by weights θ. We wish to train two modules, CCOT and DECODE, respectively parameterized by weights φ and ψ. At a high level given a query, CCOT_φ is responsible for the generation of contemplation tokens. DECODE_ψ is responsible for decoding the answer conditioned on the initial query and contemplation tokens.

Consider a training instance consisting of a query, full reasoning chain and answer, denoted as $w_{1:n}, t_{1:m}$ and $a_{1:o}$, respectively. Assume some fixed compression ratio $0 < r < 1$ and let $k = \lceil r \cdot m \rceil$. This compression ratio controls how much the reasoning chains are compressed; $r = 1$ corresponds to finetuning on the full reasoning chain while a $r = 0$ corresponds to finetuning on just the answer. φ and ψ are fine-tuned successively, each initialized from θ.

4.1. Finetuning CCOT_φ

The goal of CCOT_φ is to generate contemplation tokens. Under CCOT, these tokens are a compressed representation of a full reasoning chain, equivalent to a size k subset of the hidden states $\hat{t}_{1:m}$ produced by LM_θ. Since processing all of t and then performing a subset selection still incurs the linear cost of generating all m tokens, CCOT_φ is thus trained to **approximate a subset of precomputed hidden states**.

To achieve this, we first precompute the hidden states of the concatenated input. We next use a checkpoint of a scorer used to perform a similar subset selection from Qin et al. (2024) in order to perform the subset selection of the hidden states. This scorer is simply a linear layer that takes the embeddings from a predetermined layer T as input, and returns the indices of the selected subset. We discuss other

[1] The average reasoning chain in GSM is 1.5 times longer than their corresponding query. Reasoning chains provided by GPT o1 are hundred of times longer than their query.

methods of subset selection in Section 5.2.

$$[\bar{w}_{1:n}; \bar{t}_{1:m}; \bar{a}_{1:o}] = \text{EMBED}_\theta([w_{1:n}; t_{1:m}; a_{1:o}])$$
$$[\hat{w}_{1:n}; \hat{t}_{1:m}; \hat{a}_{1:o}] = \text{ATTN}_\theta([\bar{x}_{1:n}; \bar{t}_{1:m}; \bar{a}_{1:o}])$$
$$I = \text{SCORER}(\hat{t}^T_{1:m})$$

We have that $|I| = k$, and we can index the hidden states $z_{1:k} = \hat{w}_I$ to serve as the gold labels. We aim to generate k contemplation tokens $\hat{z}_{1:k}$ conditioned on $w_{1:n}$ under φ to approximate the labels, but is not immediately clear what inputs we should use to generate the contemplation tokens.

A reasonable choice is to use the embeddings of the tokens corresponding to the selected indices, $\hat{w}_I$. This choice would make the hidden state approximation easier due to skip connections in the attention layer: $\hat{w}_I$ are the exact inputs used to compute the hidden states in the noncompressed case. However, the selected tokens are usually punctuation tokens and articles. This choice would require predicting a random sequence of semantically empty tokens when autoregressively decoding as we pass the last layer embeddings $\hat{z}^L_i$ through the language model head. Another option would be to learn a single embedding as input to generate each hidden state, but this choice removes the additional computational depth induced by autoregressive decoding.

We instead take inspiration from reasoning over continuous space and use the intermediate hidden layers of the previous contemplation token as input to the next token. Formally, the inputs to generate the contemplation tokens $\hat{z}_{1:k}$ are the embeddings of $z^l_{0:k-1}$ at some fixed layer l where z_0 represents the hidden state of the last token of the query.

This choice is quite natural as it generalizes the naive autoregressive decoding strategy (Section 4.3). We train the parameters of φ layer by layer with the following loss:

$$\text{LOSS}_\varphi(z^l_i, \hat{z}^l_i) = \frac{1}{k}\sum_{i=1}^{k} \frac{1}{\sigma^2(z^l_i)} \text{MSE}(z^l_i, \hat{z}^l_i)$$

where $\sigma^2(z)$ denotes the variance of z and MSE denotes the usual mean squared error between two vectors. We use a scaled mean squared error in order to normalize hidden states with average L^1 norms. These norms differ drastically between different layers within the same model, so the scaled loss allows us to keep a consistent learning rate.

To train the ith layer, we pass in the inputs described above and compute forward passes through i Transformer layers, crucially only updating the parameters corresponding to the ith layer. When training subsequent layers, the parameters corresponding to the ith layer are frozen. This provides a natural segmentation to the approximation task, and we found this improved the generated contemplation tokens.

4.2. Finetuning DECODE_ψ

We assume a trained module CCOT_φ. Compressed reasoning chains are out of distribution for θ, so we need a separate module in order to effectively condition on the generated contemplation tokens. We train DECODE_ψ to **decode the answer from the query and contemplation tokens**.

To do this, we first encode the hidden states of the query and autoregressively generate contemplation tokens $z^*_{1:k}$. Con-

Algorithm 1 Chain of Thought inference

Require: Query w, parameters θ
$\bar{w} \leftarrow \text{EMBED}_\theta(w)$ ▷ *embed query*
$\hat{w} \leftarrow \text{ATTN}_\theta(\bar{w})$ ▷ *compute hidden states*
$z \leftarrow [\langle COT \rangle]$
while $z_{-1} \neq \langle ANS \rangle$ **do**
  $[\hat{w}; \hat{z}] \leftarrow \text{ATTN}_\theta([\bar{w}; \text{EMBED}_\theta(z)])$
  $x \sim \text{HEAD}_\theta(\hat{z}^L_{-1})$
  $z \leftarrow [z; x]$
end while
$a \leftarrow [\langle ANS \rangle]$
while $a_{-1} \neq \langle EOS \rangle$ **do**
  $[\hat{w}; \hat{z}; \hat{a}] \leftarrow \text{ATTN}_\theta([\bar{w}_{1:n}; \text{EMBED}_\theta([z; a])])$
  $x \sim \text{HEAD}_\theta(\hat{a}^L_{-1})$ ▷ *sample answer token*
  $a \leftarrow [a; x]$
end while
return a

Algorithm 2 CCOT inference

Require: Query w, parameters θ, φ, ψ, autoregressive layer l
$\bar{w} \leftarrow \text{EMBED}_\theta(w)$ ▷ *embed query*
$\hat{w} \leftarrow \text{ATTN}_\theta(\bar{w})$ ▷ *compute hidden states*
$z \leftarrow [\hat{w}^l_{-1}]$
while $\text{END}_\psi(\hat{z}^L)$ is $False$ **do**
  $[\hat{w}; \hat{z}] \leftarrow \text{ATTN}_{\theta,\varphi}([\bar{w}; z])$ ▷ *gen. cont. token*
  $z \leftarrow [z; \hat{z}^l_{-1}]$ ▷ *append cont. token*
end while
$a \leftarrow [\langle ANS \rangle]$
while $a_{-1} \neq \langle EOS \rangle$ **do**
  $[\hat{w}; \hat{z}; \hat{a}] \leftarrow \text{ATTN}_{\theta,\varphi,\psi}([\bar{w}_{1:n}; z; \text{EMBED}_\theta(a)])$
  $x \sim \text{HEAD}_\psi(\hat{a}^L_{-1})$ ▷ *sample answer token*
  $a \leftarrow [a : x]$
end while
return a

Figure 2. Two algorithms for inference with contemplation tokens. Algorithm 1 describes the usual chain of thought decoding while Algorithm 2 describes our method, obtained by replacing the yellow text with the cyan text. While CoT decoding decodes contemplation tokens by passing the LM head across the final hidden state, we use the hidden state at the lth layer directly.

trasting the training of CCOT_φ, we perform this generation **autoregressively** rather than using the precomputed embeddings $z^l_{0:k-1}$ described in Section 4.1 We start by passing in z^l_0 and compute the hidden states $\hat{z}_1$. We then autoregressively take $\hat{z}^l_1$ as the next input to generate $\hat{z}_2$, until an entire sequence $\hat{z}_{1:k}$ is generated. Then, conditioning on the query and contemplation tokens, we pass in the answer tokens $a_{1:o}$ and compute the next-token distributions $p_{1:o}$.

We finetune ψ with the usual cross-entropy loss given the computed distributions where the probabilities of the next token a_i are drawn from the distribution p_{i-1}.

$$\text{LOSS}_\psi(a_{1:o}) = -\sum_{i=2}^{o} \log p(a_i \mid a_{1:i-1}) \sim p_{i-1}$$

The tokens of a are conditioned on the contemplation tokens $\hat{z}$ generated under φ. By unfreezing the parameters φ when finetuning the parameters ψ using LOSS_ψ, we note that the parameters φ receive signal from the downstream task.

Empirically, we find that this signal is not entirely useful – downstream performance decreased if all the parameters φ are unfrozen. We hypothesize that updating the parameters corresponding to earlier layers affects the autoregressive generation of the contemplation tokens. As such, we find that unfreezing the parameters corresponding to layers *after* the autoregressive layer l ends up improving performance.

k will not be known during test time, so we additionally train a binary classifier END_ψ that takes the final layer of generated hidden states $\hat{z}^L_i$ as input and either predicts whether another contemplation token token should be generated. We stop generating contemplation tokens after h tokens. We set $h = 200r$, which would only prematurely terminate less than 3% of the long tailed distribution of reasoning chains.

4.3. Inference

Assume we have a pretrained causal decoder-only language model parameterized by weights θ. Additionally, assume trained modules CCOT_φ, DECODE_ψ and the end predictor END_ψ. Given a query w, we describe inference in Algorithm 2. We remark that our method to generate contemplation tokens is quite natural; Algorithm 1 describes the usual chain of thought inference and the differences are marked, cyan for our method and Yellow for naive CoT decoding.

The most crucial difference is that when CCOT generates contemplation tokens (lines 4-7), it uses lth layer of the last token's hidden state as a *continuous* next input. In contrast when CoT generates contemplation tokens, it uses the final Lth layer to do the usual autoregressive decoding described in Section 3.1, passing to a *discrete* set of tokens. Moreover, if m is the average length of reasoning chains under θ, CCOT will generate on average only $k = \lceil r \times m \rceil$ contemplation tokens, whereas CoT will generate on average all m tokens.

4.4. Implementation Details

We use LORA (Hu et al., 2021) in order to finetune φ and ψ with ranks of 128 and 64, respectively. When generating the gold hidden states, we pass the $T = 3$ layer to perform our subset selection and the $l = 15$ as inputs. We also take the hidden state at the lth layer to do autoregressive generation of contemplation tokens when finetuning ψ and during inference. We use the decoder-only Transformer architecture of LLAMA for our experiments, taking the LLAMA2-7B-CHAT checkpoint (Touvron et al., 2023) as our base model.

5. Experiments

5.1. Experimental Setups

We evaluate our CCOT framework on the reasoning dataset GSM8K (Cobbe et al., 2021). For the reasoning chains required to train both modules, we use the chains of thought provided with the dataset. We remove all calculator annotations present in the reasoning chain, only keeping the natural language reasoning. We finetune φ with precomputed gold states with two compression ratios, $r = [0.05, 0.10]$. We emphasize that the choice of r is a training time decision, CCOT_φ approximates the hidden states under the fixed compression ratio r.

We compare our results to two baselines of $r = [0.0, 1.0]$. These compression ratios are the two extreme values of the compression spectrum we introduce, corresponding to the cases of no reasoning chain and full reasoning chain. We finetune the model with the usual cross entropy loss on the dataset; For $r = 0.0$, the model directly outputs the answer without generating any contemplation tokens. For $r = 1.0$, the model generates the explicit reasoning chain as its contemplation tokens during inference.

Additionally, we compare to PAUSE, a method derived from Goyal et al. (2024). We finetune the model with no reasoning chains, but for a given ratio r, append $k = \lceil r \times m \rceil$ contemplation tokens between the query and answer where m is the length of the reasoning chain. We learn the input embedding of the special token, chosen to be $\langle pause \rangle$. These pause tokens are added to provide the model with an enhanced computational width (See Section 6.2 for further discussion). We evaluate with the same compression ratios $r = [0.05, 0.10]$ to measure the effect of the tokens.

5.2. Results and Discussion

We provide our main results in Table 2. Accuracy refers to the exact match accuracy obtained on the test set with no in-context examples. Decode time refers to the average time to generate an answer to a test set query, measured in

Format	$1/r$	Acc. (EM)	Decode Time
CCOT	∞	0.089	0.33
CCOT	20x	0.151	0.49
CCOT	10x	0.179	0.78
CCOT	1x	0.315	8.10
PAUSE	20x	0.092	0.35
PAUSE	10x	0.099	0.37

Table 2. Accuracy and decode time on GSM8K (Cobbe et al., 2021) contrasting our method, CCOT, and PAUSE (Goyal et al., 2024) each equipped with two different compression ratios against baselines of no compression (full reasoning chains) and infinite compression (no contemplation tokens). Higher accuracy indicates better performance, while lower decode time indicates better efficiency.

seconds by wall clock time on a single Nvidia A100 GPU.

With a compression ratio of $r = 0.10$, we see a 9 point improvement over the baseline with no contemplation tokens. This accuracy gain is achieved with an only 0.4 second increase in generation time. If we reduce r to 0.05, we still see a sizable 6 point improvement over the baseline, with a generation time increase of only around 0.15 seconds. In contrast, even though the contemplation tokens generated by PAUSE could be decoded faster, they were only able to nominally improve performance. We hypothesize that even though these tokens provide the model with additional computations, reasoning datasets like GSM8K require more sequential computations over parallel ones. Ultimately, our results show equipping a model with dense, contentful contemplation tokens produced by CCOT allows the model to reason better than if it had no contemplation tokens, or used a discrete set of noncontentful ones.

6. Further Discussion

6.1. Hyperparameter Choices

Varying r As r controls how many contemplation tokens are generated, it makes sense that increasing r would increase both accuracy and decode time. However, we found that accuracy plateaus after a certain threshold, about $r = 0.2$. We hypothesize that this occurs because successive contemplation tokens are autoregressively decoded using the hidden state at the l layer, which propagates an approximation to the next contemplation token generation. We suspect the noise from the approximation errors eventually outweighs the signal provided by the contemplation tokens.

Varying l We find that the choice of l is important – we were unable to learn good weights for φ when l was set close to either 0 or the last layer L. We hypothesize that hidden states at earlier layers (small l) still incorporate a lot of localized information about the token itself while hidden states at later layers (large l) instead incorporate a lot of localized information about the *next* token. As such, we found that $l \approx L/2$ resulted in the best performance; we hypothesize that the hidden states at intermediate layers encode global information it them suitable for autoregressive decoding scheme we use to generate contemplation tokens. We provide results with other layer choices in Appendix A.

Subset selection We used a learned scorer module to perform the subset selection of the gold hidden states to be emulated by φ. In practice, we found that simply taking k evenly spaced tokens resulted in a similar performance. However, we note that a module trained to decode from gold hidden states (in the setup described in Section 3.3) achieves lossless performance compared to decoding from the full reasoning chain, even for small values of r. As such, we hypothesize that it is possible to learn a better scorer to identify a subset of hidden states that is easier to emulate; a better approximation of the gold hidden states could lead to lossless performance while only taking a fraction of the time to decode. The observed performance-efficiency tradeoff also likely occurs because it is easier to approximate sequences with less compression.

6.2. Theoretical Considerations

We explore the enhanced computational expressivity offered by contemplation tokens and crucially identify the advantage of decoding contemplation tokens autoregressively rather than in parallel. We provide a few high level intuitions that are formalized in Appendix B.

Width Suppose we have a Transformer block ATTN and an input $\bar{w}_{1:n}$. Computing $\text{ATTN}(\bar{w}_{1:n})$ results in $O(n)$ parallel operations. If we pass in m additional contemplation tokens in parallel and compute $\text{ATTN}(\bar{w}_{1:n+m})$, we now perform $O(n+m)$ parallel operations. The extra computations matter in tasks when the number of parallel operations required is greater than the input sequence length. This can occur when answering succinctly phrased problems that require many parallel operations: "compute all pairwise sums in the following list" or "select all combinations of dependent logical statements that can be mutually true" Computing pairwise sums of an n element list requires processing $O(n^2)$ parallel computations and computing the validity of all possible logical combinations of n facts requires processing $O(2^n)$ ones. As n grows, introducing contemplation tokens during inference in these *width-bottlenecked* scenarios can allow models to solve additional problems.

Depth Suppose we have a model consisting of L Transformer blocks, and we generate contemplation tokens autoregressively than in parallel. Passing in m additional contemplation tokens still results in the increased $O(n+m)$ parallel

operations, but also resuts in $O(mL)$ sequential operations. These extra computations matter in tasks when the number of sequential operations required is greater than the depth of the model. This can occur in multi-hop question answering tasks or when determining the best move in sequential games such as go and chess. Introducing autoregressively decoded contemplation tokens in these *depth-bottlenecked* scenarios can allow models to solve additional problems.

To collect these observations into a formal theorem, we build from prior work that provides an analysis of the computational power of contemplation tokens decoded in parallel (Goyal et al., 2024). We restate their theorem below:

Theorem 6.1 (From Goyal et al. (2024)). *Assume that the attention module has sufficiently many parameters (K) that is much larger than the number of input tokens (N). Then there are tasks that M independent computations, where $N < M < K$), such that a 2-layer Transformer can implement the task if and only if it uses contemplation tokens.*

Under the same assumptions, autoregressively decoded contemplation tokens can solve a broader class of problems. When the depth of a task D exceeds the number of layers in a model L, the model can only represent L steps out of the required D steps. Our intuition is that contemplation tokens can "save" the intermediate steps, so autoregressively the model's representation of the Lth step as the input to the next token allows for the next forward pass to implement another L steps on top of the saved work. Thus, any task of depth D can be solved with an additional D/L contemplation tokens. We note that in CCOT, we only pass in the representation of the $l \approx L/2$ step, but this doesn't detract from the asymptotic representational capacity; we simply require an additional D/l tokens instead of D/L. We informally state our theorem below, see Theorem B.5.

Theorem 6.2. *Assume the conditions in Theorem 6.1. Then, there are tasks that involve M independent computations of a depth $D > 2$ such that a 2-layer Transformer can implement the task if and only if it autoregressively decodes contemplation tokens.*

7. Conclusion

We propose a new framework, CCOT, to generate contentful and autoregressively decoded contemplation tokens, a term we introduce to unify the terms given to tokens introducing additional computation to a language model. CCOT provides substantial improvements over the baseline as well as methods introduced by prior work. We additionally show how reasoning can be viewed as efficiency-performance tradeoff through the adaptive compression ratio. Overall, our work demonstrates the potential of using contentful contemplation tokens as an alternative to explicit reasoning chains, suggesting a new paradigm of reasoning in continuous space.

References

Burtsev, M. S., Kuratov, Y., Peganov, A., and Sapunov, G. V. Memory transformer, 2021. URL https://arxiv.org/abs/2006.11527.

Cobbe, K., Kosaraju, V., Bavarian, M., Chen, M., Jun, H., Kaiser, L., Plappert, M., Tworek, J., Hilton, J., Nakano, R., Hesse, C., and Schulman, J. Training verifiers to solve math word problems, 2021. URL https://arxiv.org/abs/2110.14168.

Deng, Y., Prasad, K., Fernandez, R., Smolensky, P., Chaudhary, V., and Shieber, S. Implicit chain of thought reasoning via knowledge distillation, 2023. URL https://arxiv.org/abs/2311.01460.

Deng, Y., Choi, Y., and Shieber, S. From explicit cot to implicit cot: Learning to internalize cot step by step, 2024. URL https://arxiv.org/abs/2405.14838.

Ge, T., Hu, J., Wang, L., Wang, X., Chen, S.-Q., and Wei, F. In-context autoencoder for context compression in a large language model, 2024. URL https://arxiv.org/abs/2307.06945.

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

Hao, S., Sukhbaatar, S., Su, D., Li, X., Hu, Z., Weston, J., and Tian, Y. Training large language models to reason in a continuous latent space, 2024. URL https://arxiv.org/abs/2412.06769.

Herel, D. and Mikolov, T. Thinking tokens for language modeling, 2024. URL https://arxiv.org/abs/2405.08644.

Hu, E. J., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Wang, L., and Chen, W. Lora: Low-rank adaptation of large language models, 2021. URL https://arxiv.org/abs/2106.09685.

Jiang, H., Wu, Q., Lin, C.-Y., Yang, Y., and Qiu, L. Llmlingua: Compressing prompts for accelerated inference of large language models, 2023. URL https://arxiv.org/abs/2310.05736.

Kojima, T., Gu, S. S., Reid, M., Matsuo, Y., and Iwasawa, Y. Large language models are zero-shot reasoners, 2023. URL https://arxiv.org/abs/2205.11916.

Kou, S., Hu, L., He, Z., Deng, Z., and Zhang, H. Cllms: Consistency large language models, 2024. URL https://arxiv.org/abs/2403.00835.

Deliberation in Latent Space via Differentiable Cache Augmentation

Luyang Liu[1], **Jonas Pfeiffer**[1], **Jiaxing Wu**[1], **Jun Xie**[1] **and Arthur Szlam**[1]
[1]Google DeepMind

Techniques enabling large language models (LLMs) to "think more" by generating and attending to intermediate reasoning steps have shown promise in solving complex problems. However, the standard approaches generate sequences of discrete tokens immediately before responding, and so they can incur significant latency costs and be challenging to optimize. In this work, we demonstrate that a frozen LLM can be augmented with an offline coprocessor that operates on the model's key-value (kv) cache. This coprocessor augments the cache with a set of latent embeddings designed to improve the fidelity of subsequent decoding. We train this coprocessor using the language modeling loss from the decoder on standard pretraining data, while keeping the decoder itself frozen. This approach enables the model to learn, in an end-to-end differentiable fashion, how to distill additional computation into its kv-cache. Because the decoder remains unchanged, the coprocessor can operate offline and asynchronously, and the language model can function normally if the coprocessor is unavailable or if a given cache is deemed not to require extra computation. We show experimentally that when a cache is augmented, the decoder achieves lower perplexity on numerous subsequent tokens. Furthermore, even without any task-specific training, our experiments demonstrate that cache augmentation consistently reduces perplexity and improves performance across a range of reasoning-intensive tasks.

Keywords: Latent reasoning, Cache augmentation, LLM

1. Introduction

Recent research (Kojima et al., 2022; Wei et al., 2022; Wu et al., 2024) has shown that enabling large language models (LLMs) to generate, or even search over, intermediate sequences of steps before producing a final answer can significantly improve performance on reasoning tasks. More broadly, providing LLMs with the ability to allocate compute adaptively during generation can lead to more effective generation within a fixed compute budget (Schuster et al., 2022). However, at a high level, many of these "extra thinking" approaches are similar in that their sequences of intermediate outputs are discrete, making them difficult to train in an end-to-end fashion, and in that their extra "thinking" (i.e. computation) is performed just-in-time, as part of the output generating process.

In this work, we introduce a fundamentally different approach, inspired by the literature on kv-cache compression (Ge et al., 2024; Mu et al., 2024). Our approach takes a step towards LLMs that can deliberate on their memories (encoded in the kv-cache), and distill these deliberations into a form usable for subsequent tasks. Specifically, our method processes the transformer's cache and augments it with a set of soft tokens produced in a single forward pass–not sequentially. This extra processing is performed by a separate model, which we refer to as a "coprocessor", while the base transformer remains frozen. Once the kv-cache is augmented with the coprocessor's output (which we term "latent embeddings"), decoding proceeds as normal until the coprocessor is called again. This approach offers the following key advantages:

End-to-end Differentiability: Our framework enables end-to-end backpropagation during coprocessor training, facilitating efficient optimization without the need for reinforcement learning techniques.

Corresponding author(s): luyangliu@google.com, aszlam@google.com

We leverage the standard language-modeling loss on pre-training data, making the method scalable. **Asynchronous Operation**: Because cache augmentation improves results many tokens beyond the augmentation point, and because the base transformer remains frozen during coprocessor training, asynchronous coprocessor operation becomes feasible. This contrasts with existing methods where additional computation occurs sequentially, and online. Our approach opens the door to models that can strategically bank computation by deliberating and refining their internal memory, independent of composing a response to a query.

We evaluate our method using Gemma-2 (Team-Gemma et al., 2024) models pretrained on a diverse dataset mixture. Our experiments demonstrate that without any fine-tuning on specific downstream tasks, our approach consistently improves performance across a range of reasoning-intensive tasks. We observed that increasing the number of injected latent embeddings generally leads to better performance. For example, we observe a 10.05% improvement on GSM8K and a 4.70% improvement on MMLU, when augmenting the Gemma-2 2B model with 64 latent embeddings. These results highlight the potential of enhancing LLMs with kv-cache coprocessing for augmenting model capabilities.

2. Methodology

We enhance a frozen LLM by training a coprocessor that inputs a key-value (kv) cache, and augments it with a set of soft tokens. This section details the architecture and training process of our approach.

2.1. Problem statement

Given an input x and a desired target output y, and a pretrained, frozen LLM parameterized by θ, we seek to learn a coprocessor, denoted by f. This coprocessor takes the kv-cache $(k_{\theta,x}, v_{\theta,x})$ generated by the frozen LLM when processing the input x as input, and outputs a sequence of latent representations z:

$$f(k_{\theta,x}, v_{\theta,x}) \longrightarrow z \tag{1}$$

The objective of learning f is to produce latent embeddings z that, when combined with the input x, improve the frozen LLM's ability to generate the correct target y. Specifically, we aim to maximize the expected log-likelihood of the target y given the input x and the learned latent embeddings z, as predicted by the frozen LLM:

$$\max E_x[\log p_\theta(y|x, z)] \tag{2}$$

2.2. Model architecture

Our proposed architecture enhances a frozen, pretrained LLM with a dedicated coprocessor module operating on the kv-cache. As illustrated in Figure 1, the interaction between these components unfolds in three stages:

- **KV-cache Generation:** The input sequence, x, is first processed by the frozen LLM to generate its corresponding kv-cache $(k_{\theta,x}, v_{\theta,x})$. This cache encapsulates the LLM's internal representations of the input. Crucially, the LLM's weights remain frozen throughout the entire process.

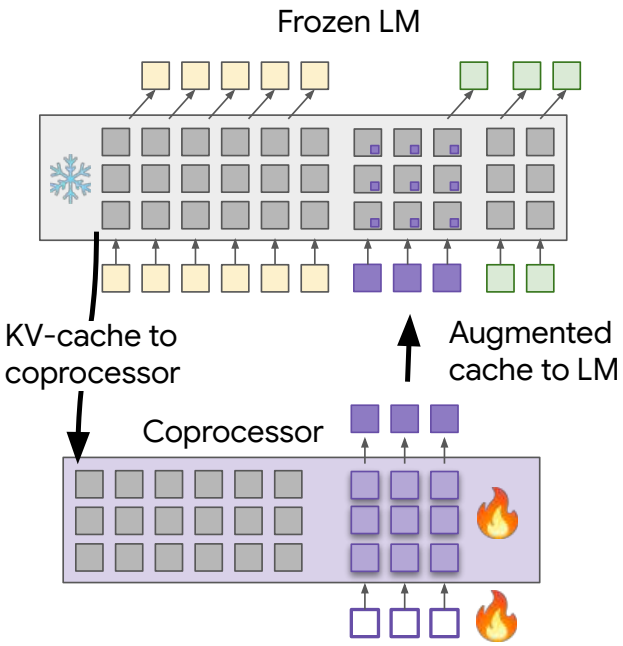


Figure 1 | Overview of the proposed architecture. The input sequence is processed by a frozen LLM, generating a kv-cache. This cache is then passed to a coprocessor, along with trainable soft tokens. The coprocessor outputs latent embeddings which are used to augment the original kv-cache before being fed back into the LLM for output generation.

- **Augmentation:** The kv-cache is then passed to the coprocessor module, which adopts the same model architecture as the pretrained LLM. The coprocessor also receives a sequence of distinct extra soft tokens with trainable embeddings. These tokens do not correspond to actual words or sub-words but serve as abstract prompts for the coprocessor. The coprocessor ingests the kv-cache and these tokens to produce a sequence of latent embeddings, z.
- **LLM Generation with Augmented Context:** Finally, z is appended to the original kv-cache. This augmented cache is then fed back into the frozen LLM, providing it with enriched contextual information derived from the coprocessor. The LLM then proceeds to generate the output sequence, y, conditioned on both the original input x and the coprocessor's output z. This allows the LLM to leverage the coprocessor's latent inferences without requiring it to explicitly verbalize intermediate steps.

Training focuses solely on optimizing the coprocessor and trainable embeddings' weights. The coprocessor shares the same model architecture as the pretrained LLM, and its weights are initialized with the pretrained weights of the LLM. The loss is calculated on the final output y, and backpropagation is used to update only the coprocessor's parameters. This targeted training approach allows for efficient fine-tuning without altering the pretrained LLM. In practice, the coprocessor's augmentation can potentially be performed offline and asynchronously, in parallel with the LLM's decoding process. This could enable continuous refinement of the LLM's contextual memory, leading to improved efficiency and faster response times.

2.3. Pretraining setup

We employ a pretraining strategy designed to encourage the coprocessor to learn augmentations that will be useful for predicting larger segments of text beyond the next token after the augmentation.

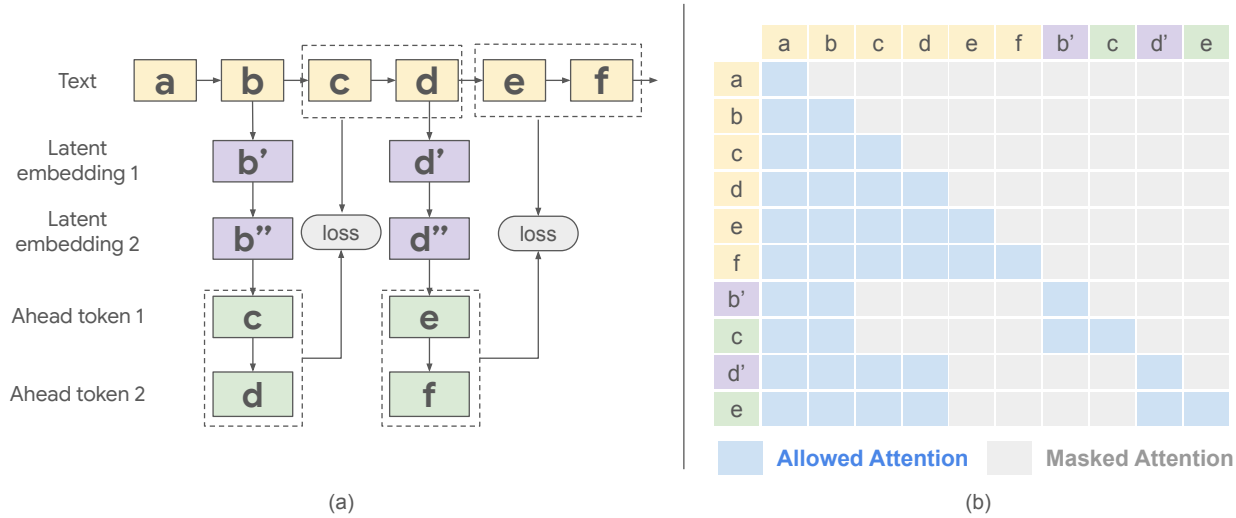


Figure 2 | Our coprocessor training framework. (a) Illustration of multi-position augmentation and ahead token prediction. For each selected augmentation position, latent embeddings are generated by the coprocessor and inserted after the corresponding token's embedding. The target tokens for prediction ("ahead tokens") are then appended. A causal mask is applied to all sequences following these insertion points. (b) Structure of the modified input and attention mask for model training. We show an example of 1 latent embedding and 1 ahead token here for simplicity.

This strategy involves generating augmentations for multiple randomly selected positions and training the coprocessor to predict many tokens ahead.

Instead of training on a single split of a sequence into input x and target y (which limits scalability), we augment at multiple points within each sequence. As shown in Figure 2 (a), given an input text sequence (e.g., "a b c d e f"), we randomly select a subset of positions (e.g., "b" and "d"). For each selected position, the coprocessor generates a configurable number of latent embeddings (e.g., b', b" and d', d" in the figure, where the number of latent embeddings is a hyperparameter N_L) in one transformer forward call. The training objective is to predict a number of tokens (another hyperparameter N_A) beyond the placement of the augmentation, in a teacher-forcing way. For instance, if "b" is chosen, and we are predicting two tokens ahead, the coprocessor uses the generated latent embeddings (b', b") and the kv-cache of the preceding text "a" and "b" to predict "c" and "d". This process can be viewed as a form of latent space interpolation: the coprocessor learns to bridge the gap between the known preceding context ("a", "b") and the future context ("c", "d") by generating meaningful latent representations (b', b"). Similarly, if "d" is chosen, the targets would be "e" and "f", based on d', d" and the preceding context "a b c d". This approach is similar to the parallel decoding introduced in Quiet-Star (Zelikman et al., 2024), but since we generate latent embeddings instead of thoughts in token space, our approach has the advantages of fully differentiability while keeping the LLM frozen. Furthermore, because the base LLM remains frozen, the coprocessor can be called asynchronously and its computations can potentially be performed in parallel with the LLM's decoding operation, and there is no need to insert between every pair of consecutive tokens.

We implement an efficient training framework by modifying the input, attention mask, position index, and target, to enable training everything together in one forward pass. Figure 2(b) illustrates how we modify the input and attention mask to enable training on multiple positions. Instead of sequentially processing each augmentation position in multiple decoder forward calls, we construct a single, extended input sequence and corresponding attention mask. This allows us to compute the loss from all selected augmentation points in parallel. For each selected augmentation position,

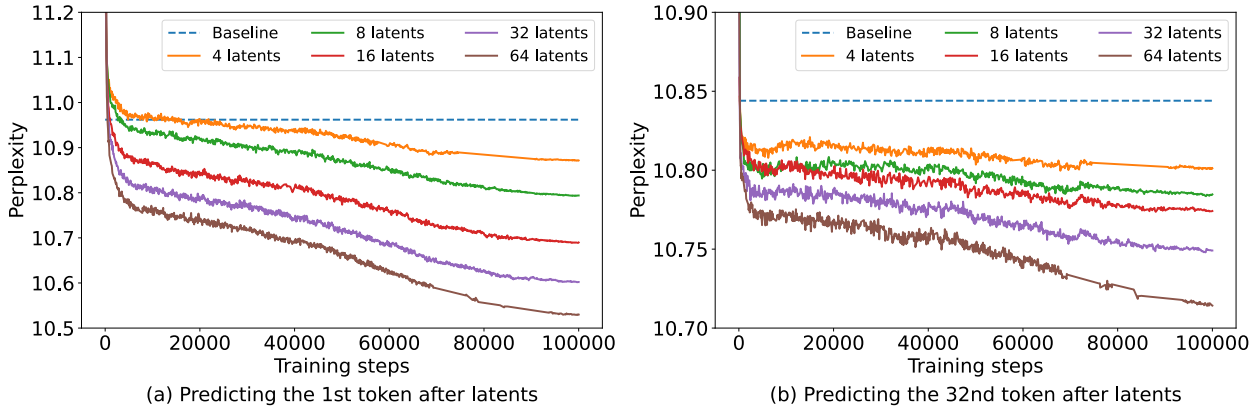


Figure 3 | Validation perplexity of the baseline frozen Gemma-2 2B model and augmented models with varying numbers of latents (8, 16, 32, 64), when predicting the 1st and 32nd tokens following latent augmentation. Lower perplexity indicates better performance.

we insert the generated latent embeddings after the corresponding token's embedding in the input sequence. The ahead tokens for that position are then appended after the latent embeddings. This creates a concatenated input sequence containing the original text followed by multiple groups of latent embeddings and their corresponding ahead tokens. The attention mask is constructed to ensure correct causal attention. The original tokens attend to all preceding original tokens, as usual. Crucially, the latent embeddings and their corresponding ahead tokens only attend to the original tokens preceding their insertion point. They are masked from attending to any subsequent original tokens, other latent embeddings, or other ahead tokens as shown in Figure 2(b). The resulting output is then used to calculate the loss and train the coprocessor.

Rather than capturing different aspects of a single token's context, these embeddings are trained to generate information useful for predicting future tokens, effectively enabling the model to perform a form of "latent thinking" before making predictions. This contrasts with standard next-token prediction and allows the coprocessor to learn more generalizable representations. By learning to anticipate future tokens in this manner, the coprocessor develops a stronger understanding of sequential dependencies within text, which proves valuable in downstream tasks.

3. Experiments

We validate our approach using the frozen Gemma-2 2B model. Our augmented Gemma-2 models, with only the coprocessor being trained and the decoder-only LLM kept frozen, are trained on the same 2 trillion token, primarily-English dataset used for Gemma-2 pretraining (Team-Gemma et al., 2024), following the setup described in Section 2.2. This dataset includes a variety of sources, such as web documents, code, and scientific articles. We trained the model for 100,000 steps using a batch size of 1024, packed sequences of length 2048, 16 ahead tokens (N_A), and 128 randomly sampled augmentation positions ("traces") for all training experiments. Importantly, no task-specific training is performed for any of the experiments; all training is done on the pretraining dataset.

3.1. Perplexity Evaluation

Our augmented Gemma model is able to achieve lower perplexity on the validation dataset compared to the pretrained Gemma model on many tokens ahead, even beyond the ahead token N_A we defined during training. We evaluate this using a proprietary validation dataset (Same as the one used in

Gemma (Team-Gemma et al., 2024)) and evaluate the effect of augmenting the frozen Gemma-2 2B LLM's kv-cache on future token prediction. For each sequence in the validation set, we generate N_L latent embeddings after each token using our coprocessor. These embeddings are then used to augment the cache at each token position. We then measure the model's ability to predict the n-th future token. Specifically, the "1st token" perplexity measures the model's performance predicting the token immediately following the inserted latent embeddings. The "32nd token" perplexity measures the model's performance predicting the token 32 positions ahead, given the context preceding the latent embeddings, the embeddings themselves, and the following 31 tokens. This demonstrates that even though we train with $N_A = 16$, the benefits of cache augmentation extend beyond this range, improving predictions even at position 32. Figure 3 presents perplexity curves during training for the baseline frozen Gemma-2 2B model and our augmented models using N_L=8, 16, 32, and 64 latent embeddings. Across all latent sizes, our approach consistently reduces perplexity, with the improvement scaling with the number of latent embeddings. This demonstrates that augmenting the cache with the coprocessor improves both short-range and longer-range prediction accuracy.

Table 1 quantifies the perplexity reduction achieved by our cache augmentation approach. The consistent reduction, generally correlating with the number of latents, confirms that the benefit of our method extends to multiple subsequent token predictions, suggesting improved internal representations within the decoder, leading to more accurate and coherent generation.

Position	8 Latents	16 Latents	32 Latents	64 Latents
1	-1.53%	-2.48%	-3.28%	-3.94%
2	-1.67%	-2.41%	-3.15%	-3.70%
4	-1.39%	-1.98%	-2.66%	-3.17%
8	-1.22%	-1.56%	-2.11%	-2.61%
16	-0.85%	-1.08%	-1.50%	-1.88%
32	-0.55%	-0.64%	-0.88%	-1.20%

Table 1 | Relative perplexity reduction (in %) achieved by augmented Gemma-2 2B models compared to the baseline, for various numbers of latents and prediction positions following latent augmentation. "Position" indicates the token position relative to the augmentation point (e.g., Position 1 is the immediately following token).

3.2. Public Benchmark Evaluation

We evaluated cache augmentation on a range of public benchmarks spanning natural language understanding and reasoning tasks (Table 2). In this setting, we only call the coprocessor *once*, at the end of the prompt. Our method consistently improves performance compared to the baseline frozen Gemma-2 2B model, with particularly substantial gains on reasoning-intensive benchmarks. Several tasks, including MMLU, GSM8K, TriviaQA, NQ, and MATH, exhibit a strong correlation between the number of latent embeddings and performance improvement. For example, on GSM8K, accuracy steadily climbs from a +1.29% gain with 4 latent embeddings to a notable +10.05% with 64. Similarly, MATH improves from -0.12% with 4 to +2.06% with 64, and MMLU shows a jump from +0.45% with 4 to +4.70% with 64. This trend suggests that for certain challenging reasoning tasks, providing more latent embeddings allows the model to perform more extensive "thinking" in the latent space, significantly enhancing its reasoning capabilities.

Other reasoning tasks, including ARC-e/c, Winogrande, and Boolq, also show improvements with increasing latent counts. While some tasks, such as AGIEval, BBH, and HumanEval, show less pronounced improvements or occasional performance dips with higher latent embedding counts, our method still frequently provides a benefit. This broad improvement across diverse benchmarks

Benchmark	Metric	Baseline	4 Latents	8 Latents	16 Latents	32 Latents	64 Latents
MMLU	5-shot	52.00	52.45 (+0.45)	52.24 (+0.24)	52.34 (+0.34)	54.61 (+2.61)	**56.70 (+4.70)**
GSM8K	8-shot	21.38	22.67 (+1.29)	23.12 (+1.74)	24.72 (+3.34)	26.76 (+5.38)	**31.43 (+10.05)**
DROP	3-shot, F1	53.69	54.64 (+0.95)	54.91 (+1.23)	56.23 (+2.55)	57.37 (+3.68)	**57.77 (+4.08)**
ARC-e	0-shot	80.56	81.52 (+0.97)	81.57 (+1.01)	83.12 (+2.57)	83.04 (+2.48)	**83.67 (+3.11)**
ARC-c	0-shot	50.26	51.28 (+1.02)	52.39 (+2.13)	53.24 (+2.99)	**54.44 (+4.18)**	**54.44 (+4.18)**
MATH	4-shot	16.50	16.38 (-0.12)	16.78 (+0.28)	17.00 (+0.50)	17.18 (+0.68)	**18.56 (+2.06)**
Winogrande	0-shot	64.01	65.35 (+1.34)	65.35 (+1.34)	66.30 (+2.29)	66.30 (+2.29)	**66.61 (+2.60)**
PIQA	0-shot	78.18	78.62 (+0.44)	78.67 (+0.49)	78.94 (+0.76)	78.94 (+0.76)	**79.00 (+0.82)**
SIQA	0-shot	51.79	51.59 (-0.20)	51.64 (-0.15)	51.74 (-0.05)	**52.30 (+0.51)**	52.00 (+0.20)
HellaSwag	0-shot	73.77	74.41 (+0.64)	74.41 (+0.64)	74.82 (+1.05)	75.04 (+1.27)	**75.31 (+1.54)**
Boolq	0-shot	75.41	75.29 (-0.12)	77.22 (+1.80)	**78.17 (+2.75)**	77.03 (+1.62)	76.91 (+1.50)
MBPP	3-shot	30.40	29.00 (-1.40)	31.60 (+1.20)	31.20 (+0.80)	31.40 (+1.00)	**31.80 (+1.40)**
AGIEval	3-5-shot	31.71	32.18 (+0.47)	30.04 (-1.67)	31.32 (-0.38)	32.78 (+1.07)	**33.85 (+2.14)**
TriviaQA	5-shot	60.29	60.30 (+0.01)	60.83 (+0.54)	61.43 (+1.14)	62.05 (+1.76)	**62.23 (+1.94)**
NQ	5-shot	17.14	17.35 (+0.21)	17.89 (+0.75)	18.16 (+1.02)	18.91 (+1.77)	**19.20 (+2.06)**
HumanEval	pass@1	19.51	18.29 (-1.22)	19.51 (+0.00)	20.73 (+1.22)	20.73 (+1.22)	**22.56 (+3.05)**
BBH	3-shot	42.22	42.36 (+0.14)	42.37 (+0.15)	42.53 (+0.31)	42.48 (+0.26)	**42.64 (+0.41)**

Table 2 | Performance of baseline and augmented models across various benchmarks. Results are shown for the baseline (frozen Gemma-2 2B pretrained model) and the model augmented with a learned coprocessor using 4, 8, 16, 32, and 64 latent embeddings, respectively. Results are reported for zero/few-shot settings as indicated in the "Metric" column. Results are accuracy (in %) if not specified in the Metric column. Improvements over the baseline are shown in parentheses. In this setting, the coprocessor is called *once*, at the end of the prompt.

underscores the effectiveness and general applicability of cache augmentation for enhancing frozen language models.

We further provide analysis in the appendix, showing that our method's performance scales with increasing training data (Section A.1) and that it effectively adapts to downstream tasks (Section A.2).

3.3. Comparison with other baselines and variations

3.3.1. Pause Token

We compare our approach with a closely related baseline: the Pause Token method (Goyal et al., 2023). Pause Token introduces trainable embeddings inserted between the input (x) and output (y) sequences, encouraging the LLM to perform latent "thinking" before generating the output. The crucial distinction between our approach and Pause Token lies in how these latent embeddings are generated. While Pause Token utilizes fixed and pretrained embeddings that do not condition on the input x, our method employs a coprocessor that generates context-dependent, dynamic embeddings based on the input. This allows our approach to tailor the latent representations to the specific input, potentially leading to more effective reasoning.

Table 3 directly compares the performance of the baseline Gemma-2 2B model against both Pause Token and our approach, with the latter two using 32 embeddings. Notably, the Pause Token model was trained using the same training data and under the same experimental setup as our method. On the validation set, our method achieves a perplexity of 10.60 on the first token prediction, significantly lower than both the baseline (10.96) and Pause Token (11.63). Furthermore, our method achieves an accuracy of 26.76% on the GSM8K dataset, outperforming both the baseline (21.38%) and Pause Token (22.37%). These improvements underscore the effectiveness of our dynamic, contextually-informed embeddings, which provide a richer representation compared to the fixed embeddings in

Method	Validation set perplexity (↓)	GSM8K 8-shot accuracy (↑)
Baseline Gemma-2 2B	10.96	21.38
Pause Token	11.63	22.37
Latent embeddings (Ours)	**10.60**	**26.76**

Table 3 | Comparison between the baseline Gemma-2 2B model, the Pause Token method (Goyal et al., 2023) (using 32 embeddings), and our approach (also using 32 embeddings). Lower perplexity indicates better next token prediction. Higher accuracy indicates better performance on GSM8K.

Baseline	0-shot CoT	16 Latents	32 Latents
21.38	23.20	24.72	**26.76**

Table 4 | Accuracy on GSM8K 8-shot for the baseline Gemma-2 2B model, zero-shot Chain-of-Thought (CoT) prompting, and our approach with 16 and 32 latent embeddings.

Pause Token, leading to better next token prediction and improved performance on reasoning tasks.

3.3.2. Zero-shot CoT

Our technique can be viewed as a form of latent Chain-of-Thought (CoT) prompting. Therefore, we compare our approach to standard zero-shot CoT (Kojima et al., 2022), which involves appending "Let's think step by step" to the input prompt. While zero-shot CoT can be effective, it relies on the LLM to generate intermediate reasoning steps token by token, which can be computationally expensive during inference. Our method, on the other hand, generates latent embeddings in a single forward pass, potentially offering a more efficient approach to guiding reasoning.

Table 4 presents the accuracy on GSM8K for the baseline Gemma-2 2B model, zero-shot CoT, and our approach with 16 and 32 latent embeddings. Our method shows clear improvements. With 16 latent embeddings, we achieve an accuracy of 24.72%, surpassing both the baseline (21.38%) and zero-shot CoT (23.20%). Performance further improves to 26.76% with 32 embeddings. This suggests that our learned, context-dependent latent embeddings provide a more efficient and effective mechanism for guiding reasoning compared to the generic prompt and sequential token generation of zero-shot CoT.

3.3.3. Alternative Coprocessor Configurations

We explored alternative configurations for our coprocessor to assess the importance of design choices. Our default setup involves finetuning the pretrained LLM to serve as the coprocessor (i.e., the coprocessor's weights are initialized with the pretrained weights of the LLM), which serves as the primary comparison point for the following experiments.

Training the Coprocessor from scratch: We also investigated training the coprocessor from scratch–randomly initializing its weights rather than finetuning from the pretrained weights of Gemma-2 2B. While training from scratch improves performance on all downstream tasks compared to the baseline, finetuning from pretrained weights yields even better results. This suggests that the coprocessor benefits from the foundational knowledge encoded in the pretrained LLM. Figure 4 illustrates this improvement in GSM8K accuracy as the number of latent embeddings increases, with the finetuned model consistently outperforming the model trained from scratch. Results of other benchmarks can be found in Table 7 in the Appendix Section A.3.

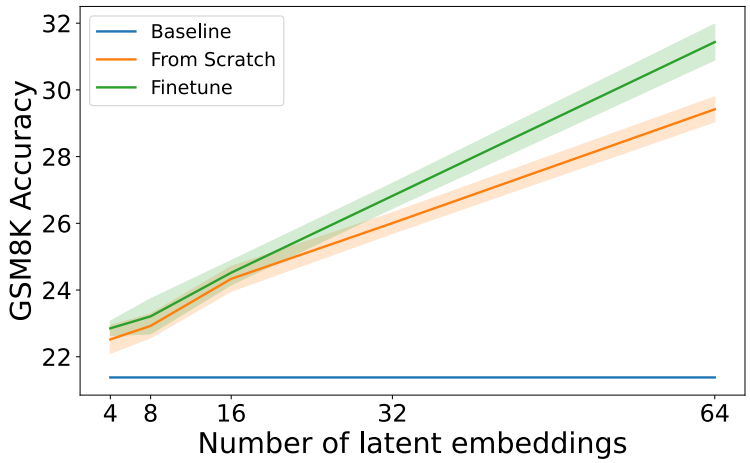


Figure 4 | Finetuning the coprocessor from Gemma-2 2B pretrained weights significantly improves GSM8K accuracy compared to training from scratch. Lines represent the mean and shaded areas represent the 95% confidence interval, both estimated from the last 5 checkpoints.

LoRA Finetuning the pretrained LLM as the Coprocessor: In addition to full finetuning the pretrained LLM and from-scratch training, we explored the efficacy of Low-Rank Adaptation (LoRA) (Hu et al., 2021) for tuning the coprocessor from the pretrained LLM's weights. LoRA freezes the pretrained model weights and introduces trainable rank-decomposition matrices, significantly reducing the number of trainable parameters. This approach offers substantial memory benefits, as only the relatively small LoRA weights need to be stored in addition to the base model. We experimented with LoRA using ranks of 64 and 128, comparing their performance on GSM8K to the baseline Gemma-2 2B model, the from-scratch training approach discussed above, and our fully finetuned coprocessor. As shown in Table 5, which presents results using 32 latent embeddings for all methods, LoRA finetuning achieves reasonable improvements over the baseline, demonstrating that even a parameter-efficient approach can effectively train the coprocessor for improved reasoning. Specifically, LoRA with rank 64 achieved an accuracy of 23.35%, while LoRA with rank 128 reached 24.03%. These results fall between the baseline performance (21.38%) and the performance achieved by from-scratch training (25.78%), indicating that while the LoRA-tuned coprocessor benefits from the pretrained weights, generating high-quality latent embeddings for effective reasoning appears to require more substantial parameter updates than those provided by parameter-efficient methods like LoRA. While these results are not as strong as the 26.76% achieved by full finetuning, they represent a notable improvement over the baseline and highlight the potential of LoRA for efficient training/inference of our coprocessor, especially in memory-constrained environments.

Method	GSM8K Accuracy
Baseline	21.38
LoRA (Rank 64)	23.35
LoRA (Rank 128)	24.03
From Scratch Training	25.78
Full Finetuning	**26.76**

Table 5 | GSM8K accuracy comparison of different finetuning methods for the coprocessor, all using 32 latent embeddings. LoRA offers a memory-efficient alternative to full finetuning, achieving reasonable performance gains.

Baseline	4 Ahead	8 Ahead	16 Ahead	32 Ahead
21.38	24.03 (+2.65)	24.11 (+2.73)	**24.72** (+3.34)	23.73 (+2.35)

Table 6 | GSM8K accuracy for varying numbers of ahead tokens during coprocessor training. 16 ahead tokens achieves the highest accuracy (24.72%, +3.34% over the baseline of 21.38%). 16 latent embeddings are used for all these experiments.

Augmentation using Last Layer's Activations: Instead of using the kv-cache as input to the coprocessor, we experimented with providing the last layer's activations from the frozen LLM, concatenated with the soft token embeddings. This approach, using 32 latent embeddings, yielded a perplexity of 10.81 on the validation set and an accuracy of 23.20% on the GSM8K benchmark under the same training setup. Both metrics are notably worse than those achieved with kv-cache augmentation, which resulted in a perplexity of 10.69 and a GSM8K accuracy of 26.76% (also with 32 latent embeddings). We hypothesize that the last layer's activations alone do not provide as rich a representation for the coprocessor as the information aggregated across multiple layers in the kv-cache, hindering both next-token prediction (reflected in the higher perplexity) and reasoning ability (reflected in the lower GSM8K accuracy).

3.4. Impact of the number of ahead token in training

We investigated the impact of varying the number of ahead tokens—the number of future tokens the model is trained to predict—during coprocessor training. While larger lookahead improves perplexity on later tokens, it often leads to higher perplexity on earlier tokens. Though learning rate scaling might mitigate this, we empirically chose 16 ahead tokens for most experiments in this paper, given its strong performance on GSM8K, as shown in the Table 6.

4. Related Work

4.1. Chain-of-Thought Reasoning in LLMs

Limitations in eliciting complex reasoning from LLMs through standard prompting have motivated research into prompting strategies that encourage intermediate reasoning steps. Chain-of-Thought (CoT) prompting (Wei et al., 2022) significantly improved reasoning performance by prompting LLMs to "think step by step". Subsequent work explored zero-shot CoT (Kojima et al., 2022; Zhou et al., 2023), aggregating multiple reasoning paths (Wang et al., 2022), internalizing the intermediate reasoning steps (Deng et al., 2024), verifying generation steps (Lightman et al., 2023), and broader search spaces for reasoning trajectories, such as Tree-of-Thought (Wang and Zhou, 2024; Yao et al., 2024). Other approaches leverage reinforcement learning (RL) to optimize the reasoning process based on final answer accuracy or target text likelihood (e.g., StaR (Zelikman et al., 2022), TRICE (Hoffman et al., 2024), Quiet-STaR (Zelikman et al., 2024)). While effective, these methods are often constrained by the expressiveness of natural language and can be computationally expensive due to the sequential generation of reasoning steps, both during training and inference.

4.2. Latent Space Reasoning

Previous research has investigated the role of latent transformer computations in LLMs' reasoning abilities. As demonstrated in (Biran et al., 2024), a sequential latent reasoning pathway in LLMs is identified for multi-hop reasoning problems. (Shalev et al., 2024) revealed that the middle layers of

LLMs produce highly interpretable embeddings, representing a set of potential intermediate answers for multi-hop queries. To improve LLMs' latent reasoning ability, researchers have proposed to augment LLMs with meta tokens. The Pause Token method (Goyal et al., 2023), closely related to our work, introduces trainable embeddings inserted between input and output sequences to encourage latent "thinking". Similarly, a recent work (Pfau et al., 2024) also studied the circumstances under which causal transformers are able to learn to utilize intermediate dummy tokens (e.g. the filler token used in this work). Unlike these studies which employ pretrained embeddings, our work generates latent tokens dynamically based on the input. More recently, COCONUT (Hao et al., 2024) introduced a new reasoning paradigm by utilizing LLMs' hidden states as input embeddings in latent space. In contrast to COCONUT, which requires multi-stage training to internalize reasoning, our approach only trains the coprocessor, thereby avoiding limitations on broader applicability.

4.3. KV-Cache Compression

KV-cache compression is a technique used to reduce the size of the transformer's kv-cache, the memory storing past activations, for efficient storage and faster computation. Ge et al. (2024) propose an in-context autoencoder (ICAE) to compress the context into concise embeddings for the kv-cache. Mu et al. (2024) introduce the concept of "gist tokens" to learn compressed representations of the kv-cache. Alternatively, prior work (Li et al., 2023) also improves the efficiency of LLMs by identifying and removing redundant information from the input, making it more compact and easier to process.

While our approach also leverages the kv-cache, our motivation is fundamentally different. Rather than focusing on compression, we aim to augment the kv-cache with latent embeddings produced by an offline coprocessor, thereby enhancing the transformer's reasoning capabilities without modifying its architecture. This allows us to improve the fidelity of further decoding and boost performance on reasoning-intensive tasks. Our work is inspired by the idea of deliberation, where the coprocessor can "think" in the latent space by processing the kv-cache and generating meaningful embeddings that guide the transformer's subsequent generation.

4.4. Augmenting LLMs with External Modules

Extensive research has focused on augmenting pretrained LLMs with external modules for improved efficiency and performance (Pfeiffer et al., 2023). Parameter-efficient fine-tuning methods like prompt tuning (Lester et al., 2021), prefix tuning (Li and Liang, 2021), and adapters have also been explored. Adapters, first introduced for computer vision by Rebuffi et al. (2017, 2018) and later popularized in NLP by Houlsby et al. (2019), insert small, trainable modules into the LLM. LoRA (Hu et al., 2021) further improves adapter efficiency by decomposing weight updates into low-rank matrices. Furthermore, multimodal models like Flamingo (Alayrac et al., 2022), CoCa (Yu et al., 2022), PaLI (Chen et al., 2022), and PaLM-E (Driess et al., 2023) leverage cross-attention or soft prompts to incorporate information from other modalities. Building upon these techniques, recent work has explored augmenting LLMs with modules specifically designed for reasoning. For example, CALM (Bansal et al., 2024) employs cross-attention between specialized and general models to enhance the general model's capabilities.

4.5. Hypernetworks for Parameter Generation

Our work shares conceptual similarities with the concept of hypernetworks, where a separate network (the hypernetwork) generates the parameters of another network. In the context of LLM augmentation, rather than learning a fixed set of parameters for a module, a hypernetwork could generate these parameters conditioned on embeddings representing different tasks, inputs, or contexts (Ha et al.,

2017; Platanios et al., 2018). This allows for a form of parameter sharing and "entanglement" between modules that are otherwise disjoint in their parameters (Goyal et al., 2021). Hypernetworks have been used to condition parameter generation on inputs as well. Examples include conditional batch normalization (de Vries et al., 2017), feature-wise linear modulation (FiLM) for text-and-vision tasks (Perez et al., 2018), and self-modulation in GANs (Chen et al., 2019). Bertinetto et al. (2016) even conditioned parameter generation on individual examples for one-shot learning. In the context of LLMs, hypernetworks have generated diverse module parameters, such as classifier heads (Ponti et al., 2021), continuous prompts (He et al., 2022), and adapter layers (Ansell et al., 2021; Mahabadi et al., 2021; Üstün et al., 2020), conditioned on task or language embeddings. These embeddings can be learned or fixed, incorporating side information about task or language relationships.

Importantly, our approach can be viewed through the lens of hypernetworks, with the coprocessor itself acting as a hypernetwork that is conditioned on the kv-cache of the frozen LLM. Instead of generating parameters for a separate module, the coprocessor generates latent embeddings that augment the kv-cache. However, the core principle remains similar: a network dynamically generating outputs based on a rich contextual input. In our case, the kv-cache serves as a highly informative representation of the input sequence, allowing the coprocessor (hypernetwork) to generate augmentations tailored to the specific context.

5. Conclusion

This paper introduces differentiable cache augmentation, a novel method for enhancing frozen decoder-only language models by incorporating a learned coprocessor that operates on the model's kv-cache. This coprocessor generates latent embeddings that enrich the context provided to the LLM, improving its ability to reason and predict future tokens without requiring any modifications to the original model architecture. Our experiments demonstrate that this approach consistently reduces perplexity and significantly improves performance on a variety of reasoning-intensive tasks, even in zero/few-shot settings. These improvements are particularly notable on tasks requiring complex reasoning, such as MMLU and GSM8K. Importantly, because the coprocessor operates offline and asynchronously, it opens up exciting possibilities for future research into models that can perform more deliberate and computationally intensive reasoning processes, including deliberation not necessarily conditioned on a responding to a particular prompt. Future work will explore scaling the coprocessor to larger models or using many modular coprocessors, investigating different coprocessor architectures, and applying this method to more diverse downstream tasks.

References

J.-B. Alayrac, J. Donahue, P. Luc, A. Miech, I. Barr, Y. Hasson, K. Lenc, A. Mensch, K. Millican, M. Reynolds, et al. Flamingo: a visual language model for few-shot learning. *Advances in Neural Information Processing Systems*, 35:23716–23736, 2022.

A. Ansell, E. M. Ponti, J. Pfeiffer, S. Ruder, G. Glavaš, I. Vulić, and A. Korhonen. MAD-G: Multilingual adapter generation for efficient cross-lingual transfer. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, pages 4762–4781, Punta Cana, Dominican Republic, Nov. 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.findings-emnlp.410. URL https://aclanthology.org/2021.findings-emnlp.410.

R. Bansal, B. Samanta, S. Dalmia, N. Gupta, S. Vashishth, S. Ganapathy, A. Bapna, P. Jain, and P. Talukdar. Llm augmented llms: Expanding capabilities through composition. *arXiv preprint arXiv:2401.02412*, 2024.

Exploring System 1 and 2 communication for latent reasoning in LLMs

Julian Coda-Forno[1,2,*], **Zhuokai Zhao**[3], **Qiang Zhang**[3], **Dipesh Tamboli**[3], **Weiwei Li**[3], **Xiangjun Fan**[3], **Lizhu Zhang**[3], **Eric Schulz**[2], **Hsiao-Ping Tseng**[3]

[1]TUM, [2]Helmholtz Munich, [3]Meta
[*]Work done during an internship at Meta

Should LLM reasoning live in a separate module, or within a single model's forward pass and representational space? We study dual-architecture latent reasoning, where a fluent Base exchanges latent messages with a Coprocessor, and test two hypotheses aimed at improving *latent communication* over Liu et al. (2024b): (H1) increase channel capacity; (H2) learn communication via joint finetuning. Under matched latent-token budgets on GPT-2 and Qwen-3, H2 is consistently strongest while H1 yields modest gains. A unified soft-embedding baseline—a single model with the same forward pass and shared representations, using the same latent-token budget—nearly matches H2 and surpasses H1, suggesting current dual designs mostly add compute rather than qualitatively improving reasoning. Across GSM8K, ProsQA, and a Countdown stress test with increasing branching factor, scaling the latent-token budget beyond small values fails to improve robustness. Latent analyses show overlapping subspaces with limited specialization, consistent with weak reasoning gains. We conclude dual-model latent reasoning remains promising in principle, but likely requires objectives and communication mechanisms that explicitly shape latent spaces for algorithmic planning.

Date: October 2, 2025
Correspondence: Julian Coda-Forno at julian.coda-forno@helmholtz-munich.de

Meta

1 Introduction

Large language models (LLMs) trained with web-scale pretraining and alignment have achieved impressive zero-shot reasoning capabilities across diverse tasks (Dubey et al., 2024; Hurst et al., 2024; Yang et al., 2025; Liu et al., 2024a). Despite their strong performance, LLMs are often seen as "fast and fluent" rather than genuinely deliberative — resembling the intuitive, System-1 side of dual-process theories of cognition (Kahneman, 2011). Indeed, recent surveys explicitly describe progress in reasoning LLMs as a shift from System-1-like heuristics to System-2-style deliberation (Li et al., 2025b), highlighting the need for architectures that support more structured reasoning.

The dominant approach today is chain-of-thought (CoT) reasoning (Wei et al., 2022; Liu et al., 2024a), where intermediate steps are verbalized in natural language. While effective, CoT incurs substantial token-level overhead, limits abstraction bandwidth, and constrains inference unnecessarily to the sequential, symbolic space of text (Chen et al., 2025; Qu et al., 2025). As model sizes and context window grow, these inefficiencies become increasingly prohibitive, motivating the search for more compact and expressive reasoning representations.

Latent reasoning (Hao et al., 2024; Liu et al., 2024b; Geiping et al., 2025) offers an alternative that enables the model to perform multi-step inference internally within its continuous hidden states, surfacing only the final answer. This paradigm promises two key advantages. First, reasoning in high-dimensional embeddings provides vastly greater expressive bandwidth than token sequences, potentially allowing richer intermediate computations (Zhang et al., 2025). Second, for combinatorial problems, operating over structured latent abstractions can dramatically reduce the effective search space (Geiping et al., 2025). Such ideas find parallels in cognitive science, where humans are believed to reason in internal "mentalese" before translating thoughts into language (Fodor, 1975).

Most latent-reasoning methods still ask a single network to do both fast association and slow deliberation;

e.g., Coconut (Hao et al., 2024) feeds the model's last hidden state back as input, forcing the same space to support next-token prediction and a putative "language of thought," creating a representational tug-of-war. Neuroscience evidence instead points to partially distinct substrates (PFC for deliberative control; striatal circuits for habitual responses (Miller and Cohen, 2001; O'Reilly and Frank, 2006; Dolan and Dayan, 2013)), suggesting that separating roles could help. KV-cache Coprocessors (Liu et al., 2024b) fit this separation, but reported gains are limited; we argue the bottleneck is *latent communication* between modules.

We therefore revisit the KV-Coprocessor design with two changes aimed at strengthening communication: (i) *frozen-Base cache augmentation*, where the Coprocessor writes cache edits that reach all layers of the Base, and (ii) *co-finetuning*, where the Base and Coprocessor are trained jointly to make the Base "listen" to latent messages. With matched token budgets and latent counts N_L, we evaluate two model families on pretraining and reasoning (GSM8K, ProsQA, Countdown) and analyze whether latents specialize. This lets us test whether these architectures deliver genuine "latent reasoning" rather than merely adding compute.

Overall, we contribute to the literature in the following ways:

1. We introduce two communication-oriented variants of Liu et al. (2024b)'s dual architecture and show better performance under matched latent-token budgets.
2. Using reasoning and latent-space analyses, we find that gains in current dual architectures largely reflect added capacity rather than structured reasoning; latents show limited specialization, suggesting that objectives encouraging specialization may be necessary for latent reasoning.

2 Related work

Reasoning in latent space. Latent-space methods shift multi-step inference from tokenized CoT into a small set of latent variables, decoupling compute from text length. Representative approaches include Coconut, which feeds continuous "thoughts" back into the model (Hao et al., 2024); differentiable KV-cache augmentation, which separates a Base token predictor from a Coprocessor that edits the cache (Liu et al., 2024b); and compressed/implicit CoT that learns dense contemplation tokens or plans without emitting text (Cheng et al., 2024; Kurtz et al., 2024). Recent surveys synthesize this trend and its claimed benefits (Li et al., 2025a; Zhu et al., 2025). Despite clear token-efficiency gains, reported improvements on *reasoning* are mixed, motivating our study of how to train dual-model systems for effective latent communication.

Communication and coordination between models. Our focus on "latent communication" between a Base and a Coprocessor connects to two strands. First, teacher–student transfer suggests that sharing an initialization can facilitate implicit trait transfer, even without explicit supervision (Cloud et al., 2025); this supports initializing the Coprocessor from the Base to align internal representations. Second, multi-agent prompting frameworks use inter-model dialogue to improve reliability (Du et al., 2023; Liang et al., 2023), though recent analyses find gains can be brittle or dataset-dependent (Wang et al., 2024). Finally, our unified *soft-embedding* baseline is grounded in continuous-prompt methods that endow a single network with extra latent capacity via trainable prefix/prompt vectors (Lester et al., 2021; Li and Liang, 2021; Goyal et al., 2023), providing a strong, parameter-efficient alternative to dual-model designs.

3 Methods

3.1 Problem setting

Setup and notation. Let B_θ be a frozen, pretrained LLM (*Base*) with parameters θ. Given a prompt x and target y, a forward pass of B_θ produces per-layer key–value caches $\{(K_\ell(x;\theta), V_\ell(x;\theta))\}_{\ell=1}^{L}$, abbreviated $KV_\theta(x)$. A Coprocessor C_ϕ with parameters ϕ reads $KV_\theta(x)$ together with N_L learnable soft tokens and emits a latent sequence $Z \in \mathbb{R}^{N_L \times d}$. At decoding time, Z is injected back into B_θ (in a manner that depends on the variant below), after which the Base generates y.

Objective. The training goal is to learn latents that improve conditional likelihood:

$$\max_{\phi \text{ (and possibly } \theta)} \mathbb{E}_{(x,y)\sim\mathcal{D}}\big[\log p_\theta\big(y \mid x, \mathrm{inject}(Z; KV_\theta(x))\big)\big],$$

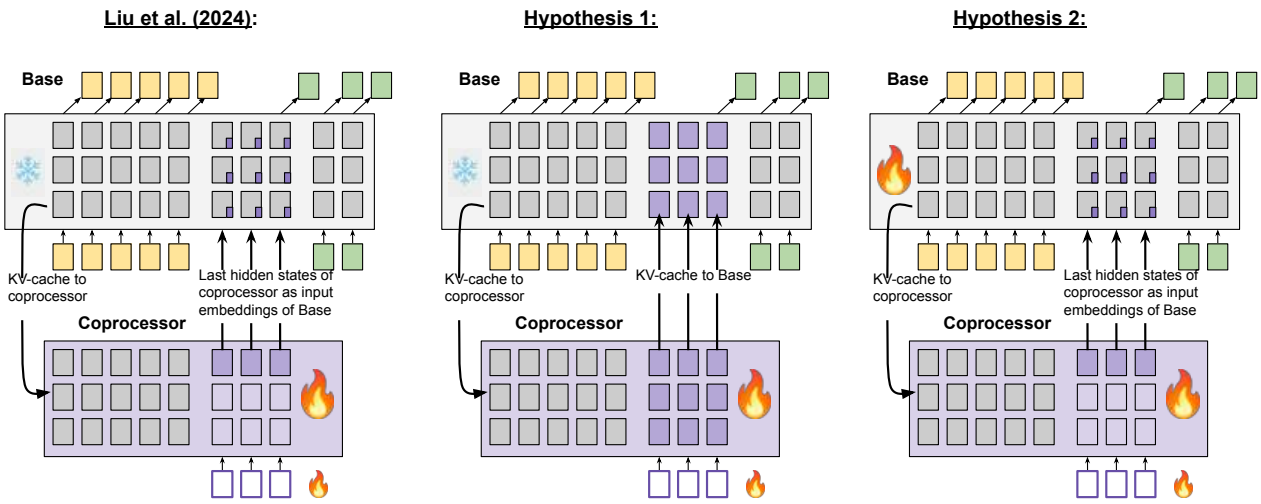


Figure 1 Overview of the deliberation–in–KV-cache architecture of Liu et al. (2024b) and our two variants designed to strengthen cross-module communication.

where $Z = C_\phi\big(KV_\theta(x), N_L\text{-soft}\big)$ and inject$(\cdot)$ denotes the chosen mechanism for feeding Z (or the Coprocessor's cache) back to the Base.

Pipeline (as in Liu et al. (2024b)). We follow the standard three-stage process (cf. Fig. 1, left):

(i) *KV-cache generation.* Run B_θ on x once to obtain $KV_\theta(x)$.

(ii) *Latent augmentation.* Run C_ϕ on $KV_\theta(x)$ plus N_L learnable soft tokens to produce a latent sequence Z. In Liu et al. (2024b), only the final layer's hidden states of C_ϕ are used as input embeddings to B_θ.

(iii) *Decoding.* Inject the augmentation ($KV_\theta(x)$ and Z) into B_θ and decode y. Unless otherwise noted, only ϕ and the soft-token embeddings receive gradients.

3.2 Experimental variants

We test two *hypotheses* aimed at strengthening communication between modules and isolating where the gains arise. Each hypothesis states a falsifiable prediction at matched latent-token budgets.

Hypothesis 1 — Frozen-Base KV augmentation. *Change:* Instead of converting the Coprocessor's last layer's output into input embeddings, we concatenate its per-layer cache to the Base cache at injection time. Let $KV_\phi(x)$ denote the Coprocessor's cache produced from $KV_\theta(x)$ and the N_L soft tokens. Decoding uses

$$[K_\ell(x;\theta)\,;\,K_\ell(x;\phi)] \quad \text{and} \quad [V_\ell(x;\theta)\,;\,V_\ell(x;\phi)] \qquad \forall\,\ell = 1,\ldots,L,$$

i.e., concatenation along the sequence (cache) dimension.[1] *Optimization:* θ remains frozen; only ϕ and the soft tokens are trained. *Motivation:* With a frozen Base, steering only via input embeddings gives the Coprocessor influence at the first hidden layer. Cache-level augmentation propagates the latent signal through *all* layers, potentially enabling richer, layer-wise "latent communication". *Prediction:* With θ frozen, cache augmentation will outperform embedding-only feedback (Liu et al., 2024b) at matched latent-token budgets N_L.

Hypothesis 2 — Co-finetuned dual-model. *Change:* Same injection mechanism as the reference system in Liu et al. (2024b) (latents as input embeddings to the Base), but we unfreeze the Base. *Optimization:* Update both θ and ϕ jointly (plus soft tokens) under the log-likelihood objective above. *Motivation:* Joint training allows B_θ to learn to "listen" to the Coprocessor's messages rather than treating them as exogenous noise. Although this increases the number of trainable parameters, it isolates whether improved cross-module interaction—rather than mere extra compute—drives downstream gains. *Prediction:* For the same latent

[1] We denote sequence-axis concatenation by $[\cdot\,;\,\cdot]$.

injection as Liu et al. (2024b), co-finetuning will outperform the frozen-Base counterpart at matched N_L, reflecting learned communication rather than added compute.

Implementation note. Both variants are realized within the same three-pass schedule used throughout the paper (§4.1): Base pass to form $KV_\theta(x) \rightarrow$ Coprocessor pass to form Z (and, for Hyp. 1, $KV_\phi(x)) \rightarrow$ Base decode with injection. This preserves a clean separation between next-token prediction and latent computation.

4 Experimental analysis:

4.1 Large scale data training

We replicate the deliberation KV-cache pipeline of Liu et al. (2024b), but adapt it to a tighter compute budget and to two model families.

Base models: GPT-2 (124 M) (Radford et al., 2019) and Qwen-3 (0.6 B) (Yang et al., 2025) replace the 2-B-parameter Gemma used in the Liu et al. (2024b) paper. This choice allows us to complete all runs within a 3-day window on an 8 × A100 (80 GB) node.

Token budget: Liu et al. train for $\approx 2 \times 10^{11}$ tokens (sequence 2048, batch 1024, 100 k steps). We scale this down to 40 B tokens for GPT-2 and 8 B tokens for Qwen-3, which we found sufficient to reproduce their qualitative trends without exceeding our hardware envelope.

Dataset. All large-scale training uses *FineWeb-Edu-100BT* (Lozhkov et al., 2024).

Sequence & latent parameters: We use sequence length S=1024, latent augmentations per sequence M=64, ahead tokens N_A=16 for back-propagating the loss into the Coprocessor (Liu et al. (2024b) use 2048/128/16), and N_L=16 latents. Figure 2 **C/D** show that validation perplexity decreases almost linearly as N_L grows under Hypothesis 2. Training cost scales with the effective per-example context $S + M \cdot (N_L + N_A)$, so larger N_L substantially raises wall-clock. Within our budget, N_L=16 provided a practical operating point, while Liu et al. (2024b) report their strongest results around N_L=32 on larger bases.

Three-pass training loop implementation (Fig. 5): Liu et al. (2024b) describe "an efficient training framework ... in one forward pass", enabled by a custom attention mask that scales to large datasets. Such a mask must support multiple augmentations (M) per sequence; otherwise only one augmentation is trained, yielding two orders of magnitude less signal per token. However, a single forward pass could allow the Base model to shortcut the intended latent computation—performing both next-token prediction and latent augmentation itself, thereby collapsing the two representational spaces and defeating the purpose of the technique. We cannot verify whether Liu et al. (2024b) avoid this, as their code is not public; the attention masks they describe imply that only one model processes the pass, but the exact implementation remains unclear. To keep our dataset large while preventing this shortcut, we adopt a strict three-pass loop (Figure 5) whose attention masks still allow M augmentations of each sequence.

Only the Coprocessor (and, in Hypothesis 2, the Base model) receives gradients. Although this three-pass schedule incurs a 3x wall-clock penalty compared with the unknown "single-pass" implementation, it preserves a clean separation between next-token prediction and latent reasoning—the property we wish to evaluate.

After convergence, models are evaluated greedily on standard pretraining benchmark for small LLMs: HellaSwag, ARC-Easy, SocialIQA, PIQA, and Winogrande.

4.1.1 Baselines

Baseline 1 - Base model + continued pretraining: Liu et al. (2024b), compare their dual-model to the initial base checkpoint, meaning the dual system has seen far more data. Instead, we keep the architecture unchanged and simply continue pretraining the base model on the **same number of tokens** used by our dual-model runs.

Baseline 2 - Base model + N_L soft embeddings: To test whether a single network can absorb the "System-2" role, we continue pretraining the base model alone while attaching the same number of new N_L learnable soft

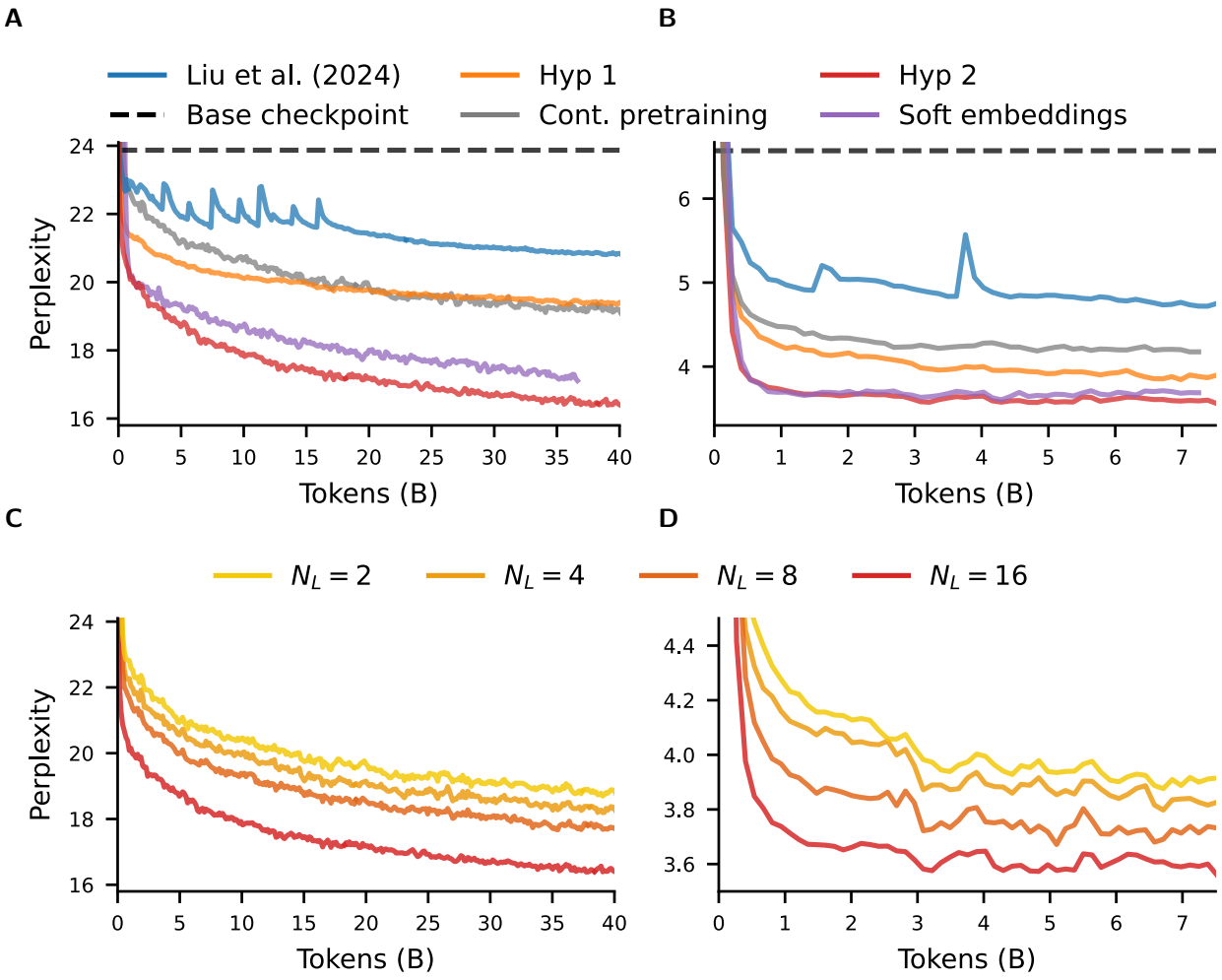


Figure 2 Validation perplexity whilst training on the FinWeb-Edu-100BT corpus. **A:** GPT-2 variants (using N_L=16 where applicable). **B:** Qwen-3 variants (using N_L=16 where applicable). **C:** Ablating number of latents N_L of the GPT-2 Coprocessor for Hypothesis 2. **D:** Ablating number of latents N_L of the Qwen-3 Coprocessor for Hypothesis 2.

tokens used by the Coprocessor setup (similar in spirit to Goyal et al. (2023), but with untied, slot-specific embeddings rather than a single shared token). This contrasts a unified versus dual-network design under identical data and latent embedding budgets. See App. A for visualization.

Note. Neither baseline is strictly parameter-matched—our dual-model has roughly twice as many weights—yet they provide useful sanity checks on data scaling and on the value of a separate Coprocessor.

4.1.2 Results

Qualitative replication of Liu et al. (2024b). Substituting the 2-B parameter Gemma with much smaller GPT-2 (124 M) and Qwen-3 (0.6 B) still produces the qualitative trends reported by Liu et al. (2024b)(Fig. 2A/B). Perplexity (ppl) drops by $\simeq 2$ for GPT-2 and $\simeq 2.5$ for Qwen-3—an order-of-magnitude larger reduction than that reported in the original paper setup. On GPT-2 the dual-model also raises mean benchmark accuracy by +2.0 percentage point (pp) (Table 1). For Qwen-3, however, lower perplexity does not translate into higher benchmark scores; continued pretraining alone even degrades accuracy. Given Qwen-3's heavy post-training, such brittleness under distribution shift is expected. We therefore report all numbers relative to the continued-pretraining baseline.

Both hypotheses outperform Liu et al. (2024b). Relative to Liu et al.'s dual-model, *Hypothesis 1* (frozen Base,

Model	Hellaswag	ARC-Easy	Social IQA	PIQA	Winogrande
GPT-2 variants					
GPT-2 continued pretraining	30.7 (+0.0)	54.2 (+0.0)	37.8 (+0.0)	64.5 (+0.0)	50.5 (+0.0)
GPT-2 + soft embeddings	30.7 (+0.0)	55.6 (+1.4)	37.7 (–0.1)	64.6 (+0.1)	51.9 (+1.4)
Liu et al. (2024b)	29.0 (–1.7)	45.0 (–9.2)	37.4 (–0.4)	62.1 (–2.4)	50.7 (+0.2)
Hypothesis 1	29.4 (–1.3)	48.2 (–6.0)	**38.7 (+0.9)**	62.9 (–1.6)	**52.3 (+1.8)**
Hypothesis 2	**31.2 (+0.5)**	**55.8 (+1.6)**	38.2 (+0.4)	**65.3 (+0.8)**	52.1 (+1.6)
Qwen variants					
Qwen continued pretraining	26.6 (+0.0)	34.5 (+0.0)	35.3 (+0.0)	55.1 (+0.0)	52.0 (+0.0)
Qwen + soft embeddings	25.9 (–0.7)	37.2 (+2.7)	34.6 (–0.7)	55.3 (+0.2)	50.4 (–1.6)
Liu et al. (2024b)	31.8 (+5.2)	36.5 (+2.0)	35.4 (+0.1)	58.1 (+3.0)	52.0 (+0.0)
Hypothesis 1	35.2 (+8.6)	45.0 (+10.5)	39.0 (+3.7)	61.0 (+5.9)	**55.7 (+3.7)**
Hypothesis 2	**37.3 (+10.7)**	**53.1 (+18.6)**	**41.9 (+6.6)**	**63.4 (+8.3)**	51.1 (–0.9)

Table 1 Model performance on standard small LLM benchmarks and Δ with continued pretraining baselines.

cache concatenation) improves average accuracy by +1.5 pp on GPT-2 and +4.4 pp on Qwen-3. *Hypothesis 2* (co-finetuned) improves by +3.7 pp (GPT-2) and +6.6 pp (Qwen-3). In ppl, the variants are ≈ 2 and ≈ 4 lower, respectively (Fig. 2A/B).

Liu model versus data-matched continued pretraining. Relative to our *data-matched* Baseline 1 (same number of training tokens), the Liu et al.'s dual-model exhibits higher ppl (≈ +1.5) and mixed accuracy: −2.7 pp on GPT-2 but +2.1 pp on Qwen-3. Averaged across families the net change is ≈ −0.3 pp, underscoring the value of stricter baselines.

Dual models versus the "soft-embedding" baseline. Our second baseline—*Base model + N_L soft embeddings*—keeps the architecture unified yet endows the network with the same latent budget. Against this control the dual-model results are underwhelming:

- Perplexity: *Hyp. 1* has higher ppl than the soft-embedding model for both families; *Hyp. 2* lowers it only marginally.
- GPT-2 benchmarks: Averaged over the five tasks in Table 1, *Hyp. 1* trails by −1.8 pp (46.3 vs. 48.1%), while *Hyp. 2* is only +0.4 pp higher (48.5%).
- Qwen-3 benchmarks: Dual models fare better: *Hyp. 1* +6.5 pp (47.2 vs. 40.7%); *Hyp. 2* +8.7 pp (49.4%).

Summary and caveat. At first glance both dual-model variants look successful: they outperform the Liu et al. (2024b) baseline and, on Qwen-3, yield sizeable benchmark gains. But the picture shifts once we introduce the *soft-embedding* control. That unified model matches the dual systems in token budget and latent capacity while using only half as many trainable weights, yet it equals or exceeds *Hyp. 1* and comes within a hair of *Hyp. 2*. This pattern indicates that the Coprocessor is not just "adding compute"—it is adding it *inefficiently*. A single LLM with the same aggregate parameter count would almost certainly do better. Accordingly, the next section turns to reasoning-specific benchmarks to see whether the dual architecture offers any benefit that a parameter-matched, soft-prompted model cannot already provide.

4.2 Reasoning evaluation: benchmarks and a controlled stress test

In this section we wanted to test if additional latent tokens improve *reasoning* or do they mostly add compute? We first compare our systems on GSM8K (Cobbe et al., 2021) and ProsQA (Hao et al., 2024), then use a controlled *Countdown* stress test (Pan et al., 2025) to probe robustness as combinatorial difficulty increases. We evaluate four systems: our reproduction of Liu et al., Hyp. 1 (frozen Base, cache-concat), Hyp. 2 (co-finetuned), and a single-model soft-embedding baseline with the same total latent budget N_L. For reference, we also compare against Coconut (GPT-2) (Hao et al., 2024). More implementation details can be found in App. B.

Benchmarks. On GSM8K/ProsQA we follow the staged curriculum of Hao et al. (2024) and report accuracy

after fine-tuning. Table 2 shows the best results ($N_L = 12$ for this task). Figure 3A plots *GSM8K* accuracy versus total latents N_L; the corresponding *ProsQA* curves are deferred to App. B, as this benchmark is near-saturated for competitive systems and adds limited insight.

Stress test: Countdown. Countdown lets us scale difficulty in a controlled, task-homogeneous way by increasing the operand count (branching factor grows rapidly with each operand). Unlike the benchmarks, we train without curriculum and sweep both $N_L \in \{1, 2, 4, 8\}$ and operands $\in \{3, 4, 5\}$. An illustrative instance appears in the box below. Figure 3B plots accuracy vs. operands, combining GPT-2 and Qwen curves for clarity.

Countdown example with operands = 4

User: Using the numbers [19, 36, 55, 7], create an equation that equals 65.
Assistant: Let me solve this step by step.
`<latent thinking>`
`<answer>` $55 + 36 - 7 - 19$ `</answer>`

Findings. (i) *Rank order on GSM8K/ProsQA:* Hyp. 2 $\geq$ soft-embedding $\gg$ Hyp. 1 $\approx$ Liu (Table 2). (ii) *Scaling latents:* Fig. 3A shows accuracy is largely *flat* as N_L increases and can dip at larger N_L. This contrasts with Section 4.1, where perplexity decreased as N_L grew, suggesting that extra latents help next-token prediction but do not translate into more reliable reasoning. Interestingly, Coconut (GPT-2) exhibits the same dipping trend, indicating the difficulty is not specific to dual-model designs. (iii) *Stress test:* In Fig. 3B, increasing operands sharply reduces accuracy for all systems; larger N_L helps slightly up to $N_L = 8$ but yields diminishing returns thereafter, and remains very close to the single-model soft-embedding baseline.

Table 2 Accuracy (%) on GSM8K and ProsQA after curriculum fine-tuning with $N_L = 12$.

	GPT-2		Qwen-3	
Model	GSM8K	ProsQA	GSM8K	ProsQA
Coconut (B0)	**34.1**	97.0	–	–
Soft emb. (B1)	26.5	97.7	38.5	**99.5**
Liu et al. (B2)	16.5	52.6	22.4	81.0
Hyp. 1 (frozen, concat)	12.0	54.1	24.0	79.0
Hyp. 2 (co-finetuned)	31.5	**99.0**	**38.6**	**99.5**

Efficiency note (Hyp. 2 vs. Coconut). Coconut generates continuous thoughts sequentially with the same network; for a given input this requires approximately N_L+1 full forward passes (each time appending a new latent and re-processing), so compute and latency scale linearly with N_L Hao et al. (2024). Our Hyp. 2 uses a strict three-pass schedule independent of N_L (Base cache $\rightarrow$ Coprocessor latents $\rightarrow$ Base decode). Assuming similar model sizes, this yields an approximate forward-pass/FLOPs reduction of

$$\text{speedup} \approx \frac{N_L + 1}{3},$$

e.g., $\sim 5.7\times$ fewer full passes at N_L=16, while remaining batch-parallel (multiple augmentations per sequence can still be handled in one Coprocessor pass). In practice this removes Coconut's serial bottleneck—while achieving comparable task accuracy on GPT-2 (GSM8K: 31.5 vs. 34.1; ProsQA: 99.0 vs. 97.0; Table 2, Fig. 3a)—enabling larger models and datasets. However, it is worth noting that Hyp. 2 also introduces roughly $2\times$ as many trainable parameters as Coconut.

Summary. Across reasoning benchmarks and the controlled stress test, adding latents mostly buys FLOPs, not robust reasoning. A unified soft-embedding model closely matches the best dual-model at equal N_L, and further increasing N_L rarely helps—sometimes hurts. This motivates the interpretability analysis in §4.3.

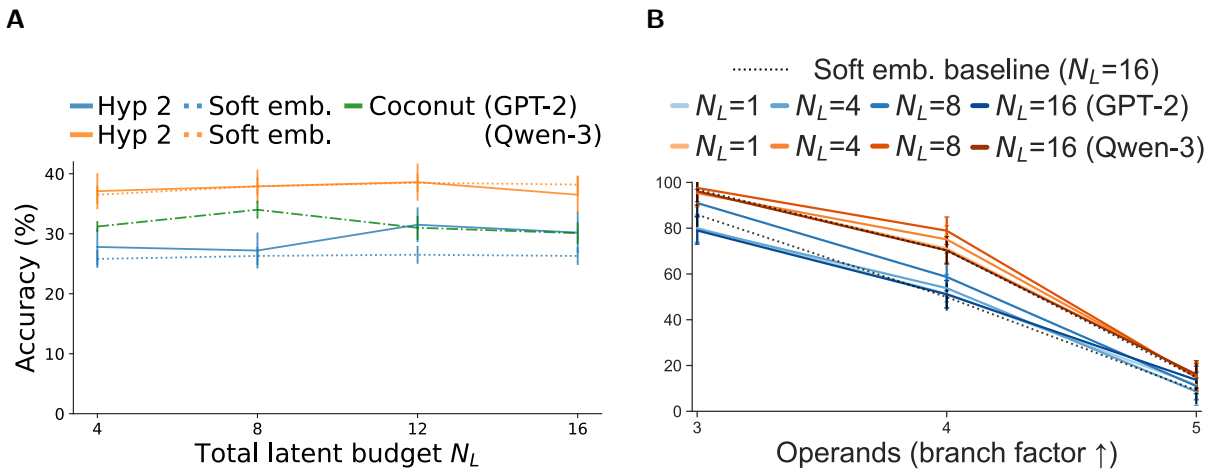


Figure 3 Ablating the latent budget. **A:** GSM8K accuracy vs. total latents N_L (GPT-2 and Qwen; Coconut shown for GPT-2). **B:** Countdown accuracy vs. operands (3, 4, 5) with lines for $N_L \in \{1, 4, 8, 16\}$, merged across model families.

4.3 Interpretability: do latents specialize or collapse?

Reasoning can be viewed as composing *distinct* intermediate computations or modules (Andreas et al., 2016; Lake and Baroni, 2017; Veličković and Blundell, 2021). If our Hypothesis 2 Coprocessor truly supports such division of labor, different *latents* should occupy meaningfully different directions in representation space. If, instead, latents mostly scale confidence without new algorithmic structure, they will reuse the same span, yielding *redundant* computations (Olah et al., 2020; Elhage et al., 2022; Kornblith et al., 2019). We therefore analyze the **Coprocessor's last hidden layer** (the signal fed back as input embeddings to the Base) and test whether latents specialize.

Diagnostic 1: Cross-capture heatmap. *Intuition.* If latents specialize, the variance of latent j's activations should not lie in latent i's subspace. Let $X_i \in \mathbb{R}^{n_i \times d}$ be row-centered activations for latent i in this layer. For each i, we fit a minimal PCA subspace explaining at least $\tau = 97\%$ of its variance and form the projector P_i. We then quantify how much of latent j's variance falls into that subspace,

$$H_{i,j} = \frac{\|X_j P_i\|_F^2}{\|X_j\|_F^2} \in [0,1],$$

visualize H as a heatmap (note $H_{i,i} \geq \tau$), and summarize with the mean off-diagonal capture

$$\overline{H}_{\text{off}} = \frac{1}{N_L(N_L-1)} \sum_{i \neq j} H_{i,j} \quad \text{(higher} \Rightarrow \text{more redundancy; lower is better).}$$

Diagnostic 2: silhouette (cluster separation by latent). We compute the silhouette score (Kaufman and Rousseeuw, 1990), $s \in [-1, 1]$, which for each point compares how close it is to its *own* latent cluster versus the *nearest other* latent cluster; higher s means tighter, better separated clusters. This complements diagnostic 1: Instead of looking at directional variance reuse across latents (orientation overlap), it looks at instance-level spatial separation/cohesion of latent-labeled clusters. We report the global average (formal definition and the mean per-latent scores in App. §C).

4.3.1 Results

Large-scale training. Figure 4a is nearly uniform and bright off the diagonal, and the quantitative scores confirm this collapse-like behavior: mean off-diagonal capture $\overline{H}_{\text{off}} = 0.9873$ (higher $\Rightarrow$ more redundancy), global silhouette $s = -0.1694$ with per-latent silhouettes mostly negative ($\{-0.26, \ldots, -0.07\}$; full vector in Appendix D). Together these indicate that occurrences labeled by different latents largely occupy the *same* subspaces and are not cluster-separated.

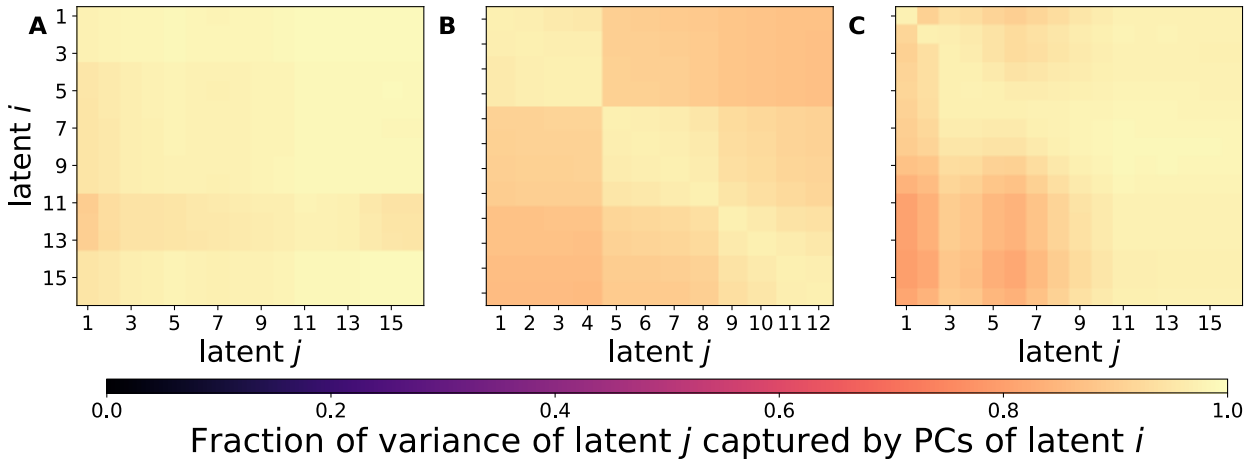


Figure 4 Latent cross-subspace capture heatmaps (last Coprocessor layer). Each cell (i, j) shows the fraction of variance of latent j captured by the principal subspace of latent i. **A:** Large-scale training. **B:** Fine-tuning on GSM8k with curriculum. **C:** Finetuning on countdown with operands = 4.

GSM8K with curriculum. Figure 4b shows the curriculum–tuned model; the no–curriculum variant is in Appendix D. Relative to the large-scale pretraining task, fine-tuning reduces *directional redundancy* among latents: $\overline{H}_{\text{off}}$ drops to 0.962 without curriculum and to 0.914 with curriculum. However, cluster separation does not emerge: the global silhouette remains near zero in both task-tuned settings ($s \approx -0.031$). In short, curriculum nudges representations in the right direction—latents become less overlapping in orientation—but they still occupy essentially the same region of space (decorrelated directions without spatial separation).

Countdown (operands = 4). Countdown induces substantially better cluster separation (global silhouette $s = 0.4531$), yet the cross-subspace redundancy remains high with $\overline{H}_{\text{off}} = 0.9382$. This realizes the "*separated but redundant spans*" regime: clusters are distinct in Euclidean space (high silhouette) but their principal subspaces still explain most of each other's variance (high off-diagonal capture). Interestingly, Fig 4c shows distinctiveness degrading beyond latent 8, aligning with Fig 3B where accuracy improves from $N_L \in \{1, 4, 8\}$ but drops at N_L=16.

Across all three settings the latents tend toward redundant computations: their subspaces are highly overlapping (high $\overline{H}_{\text{off}}$), even when the data task encourages class separation (Countdown). Task-aligned fine-tuning helps (curriculum < pretraining in $\overline{H}_{\text{off}}$), but not enough to yield "separated and specialized" latents (high silhouette and low off-diagonals).

5 Discussion

Building on Liu et al. (2024b), we cast the coprocessor architecture as a principled attempt to disentangle "System-1" token prediction from "System-2" abstract reasoning. Treating latent embeddings as a private communication channel between the two modules clarifies the conceptual link to dual-process theories in cognitive science and motivates our two new training hypotheses. However, in our setting, simply providing the channel (as in Liu et al. (2024b)) or strengthening it (our H1/H2) did not induce System-2-like computation.

Replacing last-layer hidden-state injection with full KV-cache concatenation yields modest perplexity and benchmark gains over the original design, yet falls short once stricter baselines are introduced. Its benefits appear sensitive to model family and vanish on harder reasoning tasks. Jointly updating both models produces the strongest results across all experiments, confirming that bidirectional adaptation facilitates cross-model communication. However, a *unified* network trained with the same soft-token budget narrows—sometimes erases—the gap, suggesting that current dual-system instantiations add compute inefficiently rather than unlocking qualitatively new reasoning abilities.

Our stricter baselines highlight scenarios where dual-model gains either disappear or can be matched closely by

simpler means. The Countdown experiments further show that scaling the latent budget beyond eight tokens fails to deliver systematic robustness as combinatorial complexity explodes. Our interpretability analysis seem to corroborate this by showing that learned latents largely occupy overlapping representational subspaces: extra latents mostly amplify confidence rather than add new algorithmic structure.

Our negative results do not invalidate the dual-system framework; they indicate that how the two models exchange information remains an open problem, and that current instantiations do not create conditions for System-2-like computation to emerge. Promising directions include (i) designing objectives that explicitly reward diversity or orthogonality in latent representations to encourage broader search, and (ii) developing training schedules that preserve large-scale language competence while gradually shaping latent spaces for multi-step reasoning.

Learning to Compress Prompts with Gist Tokens

Jesse Mu, Xiang Lisa Li, Noah Goodman
Stanford University
muj@cs.stanford.edu, {xlisali,ngoodman}@stanford.edu

Abstract

Prompting is the primary way to utilize the multitask capabilities of language models (LMs), but prompts occupy valuable space in the input context window, and repeatedly encoding the same prompt is computationally inefficient. Finetuning and distillation methods allow for specialization of LMs without prompting, but require retraining the model for each task. To avoid this trade-off entirely, we present *gisting*, which trains an LM to compress prompts into smaller sets of "gist" tokens which can be cached and reused for compute efficiency. Gist models can be trained with no additional cost over standard instruction finetuning by simply modifying Transformer attention masks to encourage prompt compression. On decoder (LLaMA-7B) and encoder-decoder (FLAN-T5-XXL) LMs, gisting enables up to 26x compression of prompts, resulting in up to 40% FLOPs reductions, 4.2% wall time speedups, and storage savings, all with minimal loss in output quality.

1 Introduction

Consider the prompt of a Transformer [34] language model (LM) like ChatGPT:[1]

```
You are ChatGPT, a large language model trained by OpenAI. You answer as concisely as
possible for each response (e.g. don't be verbose). It is very important that you answer
as concisely as possible, so please remember this. If you are generating a list, do not
have too many items. Keep the number of items short.
Knowledge cutoff: 2021-09 Current date: <TODAY>
```

With millions of queries a day, an unoptimized ChatGPT would encode this prompt over and over with a self-attention mechanism whose time and memory complexity is quadratic in the length of the input. Caching the Transformer activations of the prompt can prevent some recomputation, yet this strategy still incurs memory and storage costs as the number of cached prompts grows. At large scales, even small reductions in prompt length could lead to substantial compute, memory, and storage savings over time, while also letting users fit more content into an LM's limited context window.

How might we reduce the cost of this prompt? One typical approach is to finetune or distill [1, 30] the model to behave similarly to the original model without the prompt, perhaps with parameter-efficient adaptation methods [15, 16, 19]. Yet a fundamental drawback of this approach is that it requires retraining the model for each new prompt (Figure 1, bottom left).

Instead, we propose **gisting** (Figure 1, top right), which compresses arbitrary prompts into a smaller set of Transformer activations on top of virtual "gist" tokens, *a la* prefix-tuning [19]. But where prefix-tuning requires learning prefixes via gradient descent for each task, gisting adopts a meta-learning approach, where we simply predict the gist prefixes zero-shot given only the prompt, allowing for generalization to unseen instructions without any additional training. Since gist tokens are much shorter than the full prompt, gisting allows arbitrary prompts to be compressed, cached, and reused for compute efficiency.

[1] reddit.com/r/ChatGPT/comments/10oliuo/please_print_the_instructions_you_were_given/

37th Conference on Neural Information Processing Systems (NeurIPS 2023).

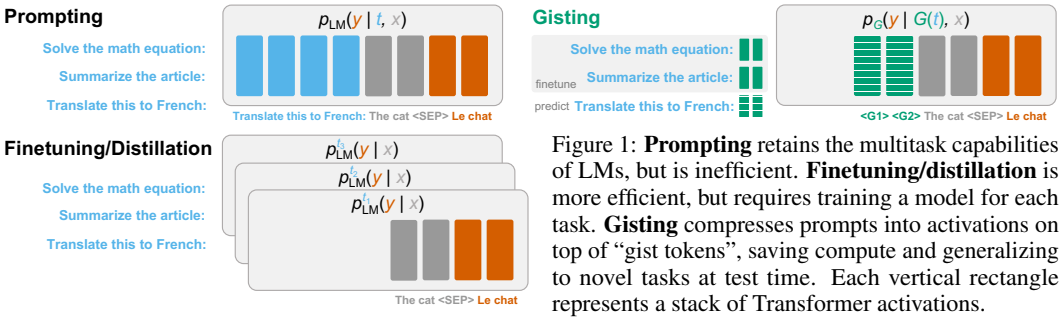


Figure 1: **Prompting** retains the multitask capabilities of LMs, but is inefficient. **Finetuning/distillation** is more efficient, but requires training a model for each task. **Gisting** compresses prompts into activations on top of "gist tokens", saving compute and generalizing to novel tasks at test time. Each vertical rectangle represents a stack of Transformer activations.

In this paper, we further propose a very simple way to learn a gist model: doing instruction tuning [38] with gist tokens inserted after the prompt, and a modified attention mask preventing tokens *after* the gist tokens from attending to tokens *before* the gist tokens. This allows a model to learn prompt compression and instruction following at the same time, with no additional training cost.

On decoder-only (LLaMA-7B) and encoder-decoder (FLAN-T5-XXL) LMs, gisting achieves prompt compression rates of up to **26x**, while maintaining output quality similar to the original models in human evaluations. This results in up to 40% FLOPs reduction and 4.2% latency speedups during inference, with greatly decreased storage costs compared to traditional prompt caching approaches.

2 Gisting

We will first describe gisting in the context of instruction finetuning [38]. We have an instruction-following dataset $\mathcal{D} = \{(t_i, x_i, y_i)\}_{i=1}^{N}$, where t is a task encoded with a natural language prompt (e.g. `Translate this to French`), x is an (optional) input for the task (e.g. `The cat`), and y is the desired output (e.g. `Le chat`). Given a (usually pretrained) LM, the aim of instruction finetuning is to learn a distribution $p_{\text{LM}}(y \mid t, x)$, typically by concatenating t and x, then having the LM autoregressively predict y. At inference time, we can *prompt* the model with a novel task t and input x, decoding from the model to obtain its prediction.

However, this pattern of concatenating t and x has drawbacks: Transformer LMs have limited context windows, bounded either by architecture or memory limits. Furthermore, given that attention scales quadratically in the length of the input, long prompts t, especially those that are repeatedly reused, are computationally inefficient. What options do we have to reduce the cost of prompting?

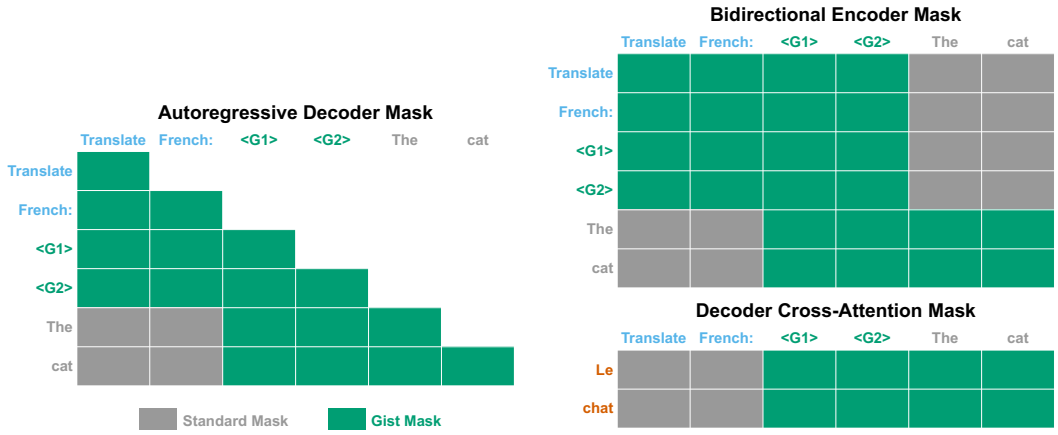
One simple option is to finetune the LM for a *specific* task t. That is, given $\mathcal{D}^t = \{(x_i, y_i)\}_{i=1}^{N^t}$, the dataset containing input/output examples only under task t, we can learn a specialized LM $p^t_{\text{LM}}(y \mid x)$ which is faster because it does not condition on t. Parameter-efficient finetuning methods such as prefix-/prompt-tuning [18, 19] or adapters [15, 16] promise to do so at a fraction of the cost of full finetuning, and newer methods like HyperTuning [25] eliminate gradient descent entirely, instead predicting the parameters of the specialized model directly from $\mathcal{D}^t$. Yet problems with these methods still remain: we must store at least a subset of model weights for each task, and more importantly, for each task t, we must collect a corresponding dataset of input/output pairs $\mathcal{D}^t$ to adapt the model.

Gisting is a different approach that amortizes both (1) the inference-time cost of prompting p_{LM} with t and (2) the train-time cost of learning a new p^t_{LM} for each t. The idea is to learn a *compressed* version of t, $G(t)$, such that inference from $p_G(y \mid G(t), x)$ is faster than $p_{\text{LM}}(y \mid t, x)$. In LM terms, $G(t)$ will be the key/value activations on top a set of *gist tokens*, smaller than the number of tokens in t, yet still inducing similar behavior from the LM. Also known as a Transformer *prefix* [19], $G(t)$ can then be cached and reused for compute efficiency. Crucially, we expect G to generalize to unseen tasks: given a new task t, we can predict and use the gist activations $G(t)$ *without any additional training*.

2.1 A Context Distillation Perspective

An alternative way to view gisting is through the lens of distillation of an already instruction-tuned LM $p_{\text{LM}}(y \mid t, x)$. Askell et al. [1] and Snell et al. [30] define *context distillation* as the process of finetuning a new LM p^t_{CD} to mimic the original LM without the prompt ("context") t, via the loss

$$\mathcal{L}_{\text{CD}}(p^t_{\text{CD}}, t) = \mathbb{E}_x \left[D_{\text{KL}}(p_{\text{LM}}(y \mid t, x) \parallel p^t_{\text{CD}}(y \mid x)) \right]. \quad (1)$$



(a) Decoder-only (e.g. GPT, LLaMA)

(b) Encoder-decoder (e.g. T5)

Figure 2: **Gist Masking**. Attention mask modifications for (a) decoder-only and (b) encoder-decoder Transformer LMs to encourage prompt compression into gist tokens <G1> <G2>. In these tables, cell (r, c) shows whether token r can attend to token c during self- or cross-attention.

The insight to be gained from this perspective is that we do not need any external data $\mathcal{D}$: this KL objective can be approximated by finetuning p_{CD}^t on a synthetic sampled dataset $\hat{\mathcal{D}}^t = \{(\hat{x}_i, \hat{y}_i)\}$ where $(\hat{x}_i, \hat{y}_i) \sim p_{\text{LM}}(\cdot \mid t)$. This is precisely the approach taken by recent work [1, 7, 30], including Wingate et al. [40], who notably learn to compress a single discrete prompt into a soft prompt via gradient descent, similar to this paper.

However, we differ from this prior work in that we are not interested in distilling just a single task, but in amortizing the cost of distillation across a *distribution* of tasks T. That is, given a task $t \sim T$, instead of obtaining the distilled model via gradient descent, we use G to simply *predict* the gist tokens ($\approx$ parameters) of the distilled model, in the style of HyperNetworks [13] and HyperTuning [25]. Our "meta" distillation objective is thus (with changes highlighted in **blue**):

$$\mathcal{L}_{\color{cyan}G}(p_{\color{cyan}G}, {\color{cyan}T}) = \mathbb{E}_{{\color{cyan}t\sim T},x}\left[D_{\text{KL}}(p_{\text{LM}}(y \mid t, x) \parallel p_{\color{cyan}G}(y \mid {\color{cyan}G(t)}, x))\right]. \tag{2}$$

In the experiments we describe below, we train on synthetic instruction-following data sampled from instruction-tuned variants of GPT-3 [3, 23]. Thus, these experiments can indeed be seen as a form of context distillation for the GPT-3 series models.

3 Learning Gisting by Masking

Having just described the general framework of gisting, here we will explore an extremely simple way of learning such a model: using the LM itself as the gist predictor G. This not only leverages the pre-existing knowledge in the LM, but also allows us to learn gisting by simply doing standard instruction finetuning while modifying the Transformer attention masks to enforce prompt compression. This means that gisting incurs *no* additional training cost on top of standard instruction finetuning!

Specifically, we add a *single* gist token g to the model vocabulary and embedding matrix. Then, given a (task, input) pair (t, x), we concatenate t and x with a set of k copies of g in between: $(t, g_1, \ldots, g_k, x)$, e.g. `Translate French: <G1> <G2> The cat`.[2] The model is then restricted such that input tokens *after* the gist tokens cannot attend to any of the prompt tokens *before* the gist tokens (but they *can* attend to the gist tokens). This forces the model to compress the prompt information into the gist prefix, since the input x (and output y) cannot attend to the prompt t.

Figure 2 illustrates the required changes. For **decoder-only** LMs such as GPT-3 [3] or LLaMA [33] that normally admit an autoregressive, causal attention mask, we simply mask out the lower-left corner of the triangle (Figure 2a). For **encoder-decoder** LMs (e.g. T5; [28]) with a bidirectional encoder followed by an autoregressive decoder, two changes are needed (Figure 2b). First, in the encoder, which normally has no masking, we prevent the input x from attending to the prompt t. But

[2] Again, the gist token is the same from g_1 to g_k; what changes is the activations on top of each token.

we must also prevent the prompt t and gist tokens g_i from attending to the input x, since otherwise the encoder learns different representations depending on the input. Finally, the decoder operates as normal, except during cross-attention, we prevent the decoder from attending to the prompt t.

Overall, these masking changes are extremely simple and can be implemented in roughly 10 source lines of code. See Appendix A for a sample PyTorch implementation which can be used as a drop-in replacement for attention masking in deep learning libraries such as Hugging Face Transformers [41].

4 Experiments

4.1 Data

A dataset with a large variety of tasks (prompts) is crucial to learn gist models that can generalize. To obtain the largest possible set of tasks for instruction finetuning, we create a dataset called Alpaca+, which combines the Self-Instruct [36] and Stanford Alpaca [31] instruction tuning datasets, each consisting of (t, x, y) tuples sampled from OpenAI's `text-davinci-001` and `text-davinci-003` variants of GPT-3, respectively. In total, Alpaca+ has 130,321 examples, with 104,664 unique tasks t, 48,530 unique inputs x, and anywhere from 0–5 inputs per task (0.64 on average).

Note that ~59% of tasks in Alpaca+ have no inputs (e.g. `Write me a poem about frogs`), in which case we simply omit the input x. While it is less interesting to cache such prompts since they are not input-dependent, they still serve as valuable training signal for learning prompt compression. Overall, while Alpaca+ is noisy and imperfect, Wang et al. [36] and Taori et al. [31] nevertheless show that models trained on such data achieve comparable performance to the original models from which the data is sampled, making this a promising testbed for studying gisting.

From Alpaca+ we hold out 3 validation splits: 1000 **Seen** prompts (with unseen, non-empty inputs); 1000 **Unseen** prompts (with non-empty inputs); and the 252 hand-written **Human** prompts and completions used in Wang et al. [36], of which 83% have non-empty inputs. The latter two splits test generalization to unseen instructions, with the **Human** split posing a stronger out-of-distribution (OOD) challenge: the average training prompt has ~20 tokens, compared to ~26 in the human split.

4.2 Models

To demonstrate gisting across multiple Transformer LM architectures, we experiment with LLaMA-7B [33], a decoder-only GPT-style model with ~7B parameters, and FLAN-T5-XXL [8], an encoder-decoder T5 model [28] with 11B parameters. For each of these models, we train models with a varying number of gist tokens $k \in \{1, 2, 5, 10\}$, using the modified attention masks described in Section 3. To assess how the model is learning prompt compression, we calibrate performance against upper- and lower-bound baselines and a simple discrete compression strategy:

Positive Control. As an upper bound on performance, we train a model with a single gist token, but without any modifications to the attention mask. This is akin to doing standard instruction finetuning.

Negative Control. As a lower bound on performance, we train a model without access to the task t. This is similar to a "random gist token" baseline, which allows us to measure how the model would do if it failed to compress *any* information into the gist prefix.

Discrete Compression with TF-IDF. An alternative approach to compression is simply using fewer discrete tokens to express the same task. Achieving compression rates similar to gisting, however, requires compression far beyond any threshold of fluency. Nevertheless, as a baseline, we compute TF-IDF statistics over the set of instructions in the Alpaca+ training set to extract the most relevant keyword in each instruction. Some examples from the training set include (see Appendix G for more):

`Write a letter to your boss asking for an increase in salary` → **`salary`**
`Given two integers, find their average` → **`average`**

We then replace each instruction in Alpaca+ with the first subword token from each keyword, resulting in compression rates equivalent to a model trained with a single gist token. Similarly to the positive control, we do standard instruction finetuning over Alpaca+ with these compressed instructions.

For full training, data, and compute details, and a link to code, see Appendix B.

4.3 Evaluation

Our evaluation uses a combination of automated metrics and AI- and human-assisted evaluation:

ROUGE-L. We first use ROUGE-L, a simple lexical overlap statistic [20], used in previous open-ended instruction finetuning work [37, 38]. The `text-davinci-{001,003}` completions are used as references, except for the Human split, where we use the gold-standard human reference.

ChatGPT. Next, we use ChatGPT-3.5 [22] to compare the outputs of our models to the positive control. While this is an imperfect metric, it allows for much faster and cheaper evaluation than human experiments, with an arguably more meaningful semantic signal than ROUGE-L. Recent work has found that ChatGPT can be used for text annotation and evaluation [12, 17, 35] with near-human performance, and similar model-based evaluations have been conducted with recent LMs [5, 11, 31].

Specifically, given a task t, input x, and outputs from two models (y_1, y_2) identified only as Assistants A and B, ChatGPT was asked to choose which assistant response is better, explaining its reasoning in Chain-of-Thought fashion [39]. If the models produced the same output, or were equally bad, ChatGPT was allowed to call a tie. We gave examples of desired outputs in ChatGPT's prompt, and randomized the order of presentation between the models for each query to avoid order effects. The full prompt given to ChatGPT and evaluation details are in Appendix C. Using these outputs, we measure the win rate of a model against the positive control: a win rate of 50% indicates that the model is of comparable quality to a model that does no prompt compression.

Human eval. Finally, after prototyping with ChatGPT, we select the best gist compression models and do a Human evaluation on a random subset of 100 of the 252 examples in the Human validation split. For each of the 100 examples, we recruited 3 US or UK-based, English-fluent annotators from Prolific, and asked them to rate model outputs in the same style as the ChatGPT evaluation above (see Appendix D for full details, including the annotation interface). The only difference is that human participants were allowed to select "I Don't Know" in cases where they had inadequate domain knowledge to accurately judge the responses, e.g. if the question was a coding question; we drop these responses (~10%) during analysis. With this human evaluation, we are not only interested in evaluating our final models, but also validating whether ChatGPT can be used as a reliable replacement for human annotation on this task.

5 Results

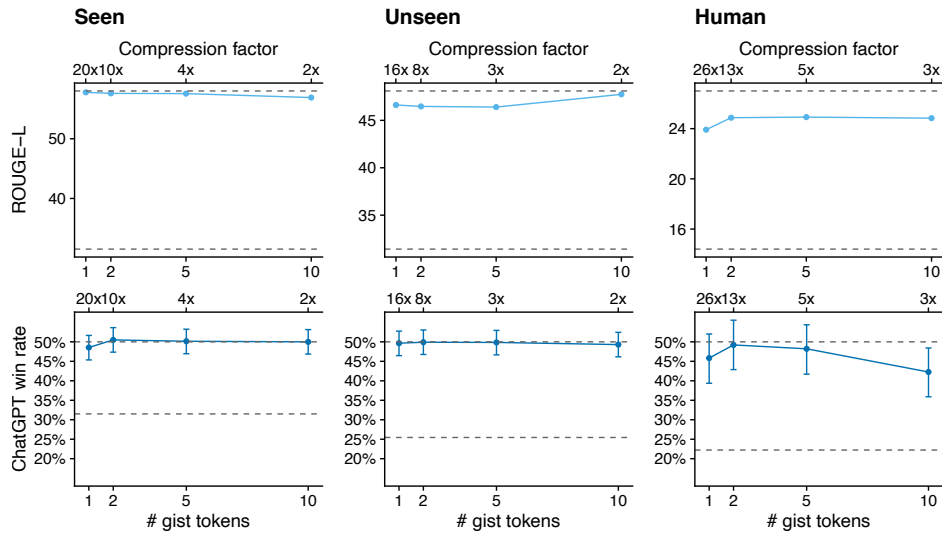
ROUGE-L and ChatGPT evaluations for LLaMA-7B and FLAN-T5-XXL, with varying numbers of gist tokens, are shown in Figure 3. Models were generally insensitive to the number of gist tokens k: compressing prompts into a single token prefix did not substantially underperform larger prefixes. In fact, having too many gist tokens hurts performance in some cases (e.g. LLaMA-7B, 10 gist tokens), perhaps because the increased capacity enables overfitting to the training distribution. Thus, we use the single gist token models for the rest of the experiments in the paper, and report the exact numbers for the single-token models, with the positive, negative, and TF-IDF baselines, in Table 1.

On **Seen** instructions, gist models attain near-identical ROUGE and ChatGPT performance as their positive control models (48.6% and 50.8% win rates for LLaMA-7B and FLAN-T5-XXL, respectively). But we are most interested in generalization to unseen tasks, as measured by the other two splits. On **Unseen** prompts within the Alpaca+ distribution, we again see competitive performance: 49.7% (LLaMA) and 46.2% (FLAN-T5) win rates against the positive controls. It is on the most challenging OOD **Human** split where we see slight drops in win rate to 45.8% (LLaMA) and 42.5% (FLAN-T5), though these numbers are still quite competitive with the positive control. Finally, gist compression is vastly superior to discrete compression; the TF-IDF models in Table 1 only marginally outperform the negative control models across the board.

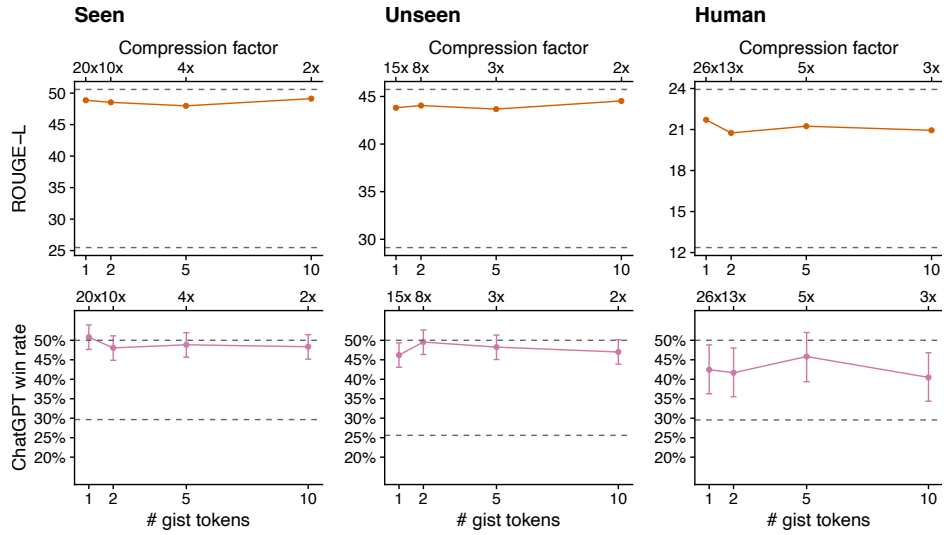
Table 2 shows the human evaluation results on the Human validation split, comparing the single gist token models to the positive control. Overall, human annotators agree with ChatGPT, with average win rates of 52.3% (vs. 48.0%) for LLaMA-7B and 40.6% (vs. 42.0%) for FLAN-T5-XXL. Importantly, this agreement persists at the level of individual responses. The average pairwise Cohen's κ among human annotators is .24 for LLaMA-7B and .33 for FLAN-T5-XXL. Because humans will often arbitrarily choose one response over another even for samples of equal quality, these numbers

Table 1: **Results for single gist tokens.** ROUGE-L and ChatGPT scores for Gist, TF-IDF, and positive/negative controls. Parentheses are scores normalized between positive/negative controls.

Model		Seen		Unseen		Human	
		ROUGE-L	ChatGPT %	ROUGE-L	ChatGPT %	ROUGE-L	ChatGPT %
LLaMA-7B	Pos	58.0 (100)	50.0 (100)	48.1 (100)	50.0 (100)	27.0 (100)	50.0 (100)
	Gist	57.8 (99.2)	48.6 (92.4)	46.6 (91.0)	49.7 (98.8)	23.9 (75.4)	45.8 (84.9)
	TF-IDF	38.1 (24.5)	34.5 (16.2)	34.0 (15.6)	29.3 (15.9)	16.5 (16.7)	24.6 (8.6)
	Neg	31.5 (0)	31.5 (0)	31.4 (0)	25.4 (0)	14.4 (0)	22.2 (0)
FLAN-T5-XXL	Pos	50.6 (100)	50.0 (100)	45.7 (100)	50.0 (100)	23.9 (100)	50.0 (100)
	Gist	48.9 (93.2)	50.8 (103.9)	43.8 (88.6)	46.2 (84.4)	21.7 (80.9)	42.5 (63.2)
	TF-IDF	32.0 (25.9)	35.9 (30.5)	34.3 (31.3)	31.0 (22.1)	13.5 (9.6)	28.4 (-5.9)
	Neg	25.5 (0)	29.7 (0)	29.1 (0)	25.6 (0)	12.4 (0)	29.6 (0)



(a) **LLaMA-7B**



(b) **FLAN-T5-XXL**

Figure 3: **Varying the number of gist tokens**. ROUGE-L and ChatGPT scores for (a) **LLaMA-7B** and (b) **FLAN-T5-XXL** for different gist tokens. Dashed lines indicate positive and negative control performance. Error bars are 95% exact binomial confidence intervals, splitting ties equally between models [10] and rounding down in favor of the positive control in case of an odd number of ties. Compression factors are calculated by computing the average token length of the validation split and dividing by the number of gist tokens.

Table 2: **Human evaluation results.** Win rate and inter-annotator agreement of single token gist models over positive control according to 3 Human annotators (H1–H3), their average, and ChatGPT (95% confidence intervals in parentheses), for 100 out of 252 examples in the Human validation split.

	Gist Win % over Pos					Agreement (Cohen's κ)	
Model	H1	H2	H3	**Human (H1–H3)**	**ChatGPT**	**Human**	**ChatGPT**
LLaMA-7B	51.1	44.5	59.8	52.3 (46.1, 58.4)	48.0 (38.0, 58.2)	.24	.29
FLAN-T5-XXL	43.0	41.9	37.2	40.6 (34.6, 46.8)	42.0 (32.2, 52.3)	.33	.29

are fairly low; however, ChatGPT shows similar levels of agreement, with average κ across each of the 3 human annotators at .29 for both models. These results, paired with the similar overall win rates, show that using ChatGPT is similar to simply recruiting an additional human annotator, and corroborates the broader results in Figure 3. See Appendix D for more human evaluation results, including a breakdown of agreement across annotators, and Appendix G for examples of instructions, model outputs, and human/ChatGPT judgments in the Human validation split.

Since our aim is having the gist models mimic the original models, one might ask how often the gist model is identical to the positive control. Figure A.3 in Appendix E shows how often this happens: for **Seen** tasks (but unseen inputs), the gist model outputs exactly match the positive control nearly 50% of the time. This drops to ~20–25% for **Unseen** tasks and ~10% for the OOD **Human** tasks.

Overall, our results show that gist models can reliably compress prompts, even to some degree those that lie outside the training distribution, especially for decoder-only LMs (LLaMA). Encoder-decoder LMs like FLAN-T5 show slightly worse OOD performance; one possible reason is that gist masking inhibits the bidirectional attention flow in the encoder, which is more challenging to adjust to than simply limiting the history of a decoder LM. Future work will be needed to investigate this hypothesis.

5.1 Failure cases

While the responses from the gist models are often indistinguishable from the positive control, we nevertheless do see failure cases where the gist token models do not perfectly compress the prompt. We give two examples where this occurs in the Human validation split: first, when the instruction contains **specific details**, e.g. phrases that must be copied verbatim into the output (in the snippets below, the names of Assistants A/B are hidden to ChatGPT):

```
Instruction: Give the news title a category. Pick a category from the list of News &
Buzz, Travel, Style, Arts & Culture, Politics, Tech, and Science & Health. <G1>
Input: The #Banksy Exhibit in Cambridge, MA is absolutely terrific.
Assistant A (FLAN-T5-XXL gist): Culture
Assistant B (pos control): Arts & Culture

ChatGPT choice: B
ChatGPT reason: The news title is about an art exhibit, so Assistant B's response of
'Arts & Culture' is more accurate and specific than Assistant A's 'Culture'.
```

We also see more cases of **runaway generations** that do not occur with the positive control model:

```
Instruction: Provide an example of how a table of contents can be generated automatically
in a LATEX document. Make sure your example contains the necessary commands. <G1>
Assistant A (pos control): \tableofcontents \chapter{Introduction} \chapter{Theory}
\chapter {Experiments} \chapter{Conclusion}
Assistant B (LLaMA-7B gist): \tableofcontents \chapter{Introduction} \section{The Basics}
\subsection {The Basics} \subsubsection{The Basics} \paragraph{The Basics}
\subparagraph {The Basics} [repeats for hundreds of tokens] \subparagraph{The Basics}

ChatGPT choice: A
ChatGPT reason: Assistant A provides a simple and correct example of generating a table
of contents in LaTeX. Assistant B's example is unnecessarily long and does not follow
the standard structure of a table of contents.
```

While it is unclear why only the gist models exhibit this behavior, these issues can likely be mitigated with more careful sampling techniques.

Table 3: **Gist efficiency improvements**. For different caching strategies (**None**, **Instruction**, **Gist**), we record CUDA wall time and GFLOPs (± std dev). Then we report the absolute/relative improvement of **Gist Caching** over these alternative strategies, with 95% confidence intervals in parentheses.

Model	Metric	Caching Strategy None	Caching Strategy Instruction[i]	Caching Strategy Gist[ii]	Absolute/Relative Δ vs. None	Absolute/Relative Δ vs. Instruction
LLaMA-7B	CUDA time (ms) ↓	23.4 ± 6.88	22.1 ± 6.58	**21.8 ± 6.55**	1.60 (1.29, 1.90) 6.8% (5.5, 8.1)	.221 (.140, .302) 1.0% (.63, 1.4)
	GFLOPs ↓	915 ± 936	553 ± 900	**552 ± 899**	362 (337, 387) 40% (37, 42)	.607 (.448, .766) .11% (.08, .14)
FLAN-T5-XXL	CUDA time (ms) ↓	31.0 ± 5.31	N/A	**29.7 ± 5.07**	1.30 (1.10, 1.51) 4.2% (3.6, 4.9)	N/A
	GFLOPs ↓	716 ± 717	N/A	**427 ± 684**	289 (268, 310) 40% (37, 43)	N/A

[i]Average KV Cache Length = 26 [ii]Average KV Cache Length = 1

6 Compute, Memory, and Storage Efficiency

Finally, we return to one of the central motivations of this paper: what kind of efficiency gains does gisting enable? To answer this question, we compare the compute requirements (CUDA wall time and FLOPs) during inference with the single-token gist models using different caching strategies:

1. **No caching**: just encoding the full prompt t.
2. **Instruction caching**: caching the activations of the uncompressed instruction t (keys and values for all layers) into what is called the **KV cache**. This is the most common caching behavior for Transformer inference [4, 26] and is supported in libraries like Hugging Face Transformers [41]. However, it is only applicable to *decoder-only* models, since in models with bidirectional encoders like T5, the instruction representations t depend on the input x.
3. **Gist caching**: Compressing the prompt into the gist prefix $G(t)$.

Table 3 displays the results of profiling a single forward pass through the model (i.e. one step of decoding with a single input token) with PyTorch [24] 2.0, averaged across the 252 Human instructions. Gist caching improves significantly over unoptimized models, with 40% FLOPs savings and 4-7% lower wall time for both models. Note that at these (relatively) small scales, the wall time improvements are smaller than the FLOPs reductions because much of the inference latency is caused by moving tensors from high-bandwidth memory (HBM) to the chip compute cores, i.e. what Pope et al. [26] call the "memory time". Larger sequence lengths and batch sizes will lead to additional speedups, as the overall latency becomes dominated by the actual matrix computations.

For LLaMA-7B, the picture is more nuanced when compared to caching the full instruction. Compute improvements of gist caching are smaller: a negligible decrease in FLOPs (0.11%) and a modest 1% speedup in wall time. This is because the FLOPs required for a Transformer forward pass is dominated by processing of the new input tokens, rather than self-attention with the KV cache. For example, a forward pass through LLaMA-7B with a single input token and a *2000-length* KV cache is only ~10% more expensive than the same forward pass with no KV cache—see Appendix F for more details. Nevertheless, this small decrease in FLOPs leads to a disproportionate decrease in wall time (1%), likely because the self-attention computations are slower relative to their FLOPs contribution.

At large scales and with heavily reused prompts, a 1% latency speedup can still accumulate into significant cost savings over time. More importantly, however, there are key benefits of gist caching over instruction caching besides latency: compressing 26 tokens into 1 gives more space in the input context window, which is bounded by absolute position embeddings or GPU VRAM. For example, for LLaMA-7B, each token in the KV cache requires 1.05 MB storage.[3] While the total contribution of the KV cache relative to the memory needed for LLaMA-7B inference is negligible at the prompt lengths we tested, an increasingly common scenario is developers caching many prompts across a large number of users, where storage costs quickly add up. In these scenarios, gisting allows caching of up to **26x** more prompts than full instruction caching, using the same amount of storage!

[3]4 (fp32 bytes) × 2 (keys+values) × 32 (num layers) × 32 (num attn heads) × 128 (head dim) = 1.05 MB.

7 Additional Related Work

Gisting builds upon past work in (parameter-efficient) instruction finetuning and context distillation, as discussed in Section 2. Here we will outline some additional connections to related work:

Adapting LMs without Backprop. As mentioned in Section 2, gisting can be viewed as a way to adapt LMs *without* gradient descent, by predicting the the prefix ($\approx$ parameters) of the adapted model. Similarly, HyperTuning [25] predicts the prefix of a model for a task using (input, output) pairs for that task. If HyperTuning is a "few-shot" adaptation method, predicting the prefix from few-shot examples, then gisting can be seen as a "zero-shot" version, predicting the prefix from the language instruction alone. Gisting also has the additional benefit over HyperTuning of being conceptually simpler: instead of training a separate LM to predict the prefix, the LM itself is used as the HyperNetwork [13], with only a tiny change to the attention masks needed for prompt compression.

Compression and memory in transformers. The idea of "compressing" prompts is closely related to previous attempts at storing past representations to improve memory and long-range sequence modeling in Transformers [9, 21, 27, 42, 43]. In particular, the Compressive Transformer [27] compresses transformer activations into a smaller *compressed memory* using a learned convolutional operator. Gisting can be seen as a variant of the Compressive Transformer with 3 key differences. First, the compression function is not a separately learned function, but the LM's own self-attention mechanism, controlled by an input-dependent gist token. Second, the compression function is learned jointly with instruction finetuning via the standard language modeling loss, not a specialized auxiliary reconstruction loss as in [27]. Finally, our task of interest is not long-range sequence modeling, but caching and reusing instruction following prompts for efficiency reasons.

Sparse attention mechanisms. By restricting attention masks, gisting draws inspiration from efficient/sparse attention methods in Transformers (see [32] for review). For example, some sliding window attention mechanisms [2, 6] may remove the need to keep the entire KV cache around during decoding, but these more general methods are not optimized for caching arbitrary parts of the input sequence of varying length, which prompt compression demands. In light of this, gisting can be viewed as an input-dependent sparse attention mechanism specifically aimed at improving efficiency of the prompting workflow now commonly used in LMs.

8 Discussion and Limitations

In this paper we presented gisting, a framework for prompt compression in LMs, and a simple way of implementing gist models by modifying Transformer attention masks that incurs no additional cost over standard instruction finetuning. Gisting can be seen either as a modified form of instruction finetuning or a method for (meta-)context distillation of an LM. Gist models can compress unseen OOD prompts up to 26x while maintaining output quality, resulting in up to 40% FLOPs reduction and 4.2% wall clock speedups over unoptimized models, and enabling new methods for prompt caching in encoder-decoder models. While wall-time improvements for decoder-only LMs are smaller, gisting nevertheless enables caching 1 *order of magnitude* (26x) more prompts relative to full instructions.

Gisting is a promising method for improving LM efficiency, but carries some limitations. While gisting seems to succeed in capturing the "gist" of instructions (hence the name), achieving such compression necessarily results in some loss of nuance of the original instruction; Secton 5.1 illustrates a concrete failure case we observed. Since the behavior of LMs on edge cases is already not well understood, it is *especially* important for practitioners to carefully evaluate whether the compute/accuracy tradeoff of gisting is sufficiently safe and robust for their use case, *before* deployment.

Nevertheless, we believe gisting opens several interesting directions for future work. First, the masking method presented here can be easily integrated into existing instruction finetuning workflows, but another exciting approach is to retrofit an existing, frozen LM by training a *smaller* model to compress prompts, if finetuning the larger LM is inconvenient. Second, the largest efficiency gains from gisting will result from compressing very long prompts, for example k-shot prompts for large k that may even exceed a single context window. Finally, compression performance can likely be improved through "gist pretraining": first learning to compress arbitrary spans of natural language, before then learning prompt compression. Such objectives could be devised by inserting gist tokens into other pretraining objectives, perhaps during language modeling or T5's span corruption objective.

The Thirty-Ninth AAAI Conference on Artificial Intelligence (AAAI-25)

IAA: Inner-Adaptor Architecture Empowers Frozen Large Language Model with Multimodal Capabilities

Bin Wang*, Chunyu Xie*, Dawei Leng†, Yuhui Yin

360 AI Research

{wangbin10, xiechunyu, lengdawei, yinyuhui}@360.cn

Abstract

In the field of multimodal large language models (MLLMs), common methods typically involve unfreezing the language model during training to foster profound visual understanding. However, the fine-tuning of such models with vision-language data often leads to a diminution of their natural language processing (NLP) capabilities. To avoid this performance degradation, a straightforward solution is to freeze the language model while developing multimodal competencies. Unfortunately, previous works have not attained satisfactory outcomes. Building on the strategy of freezing the language model, we conduct thorough structural exploration and introduce the Inner-Adaptor Architecture (IAA). Specifically, the architecture incorporates multiple multimodal adaptors at varying depths within the large language model to facilitate direct interaction with the inherently text-oriented transformer layers, thereby enabling the frozen language model to acquire multimodal capabilities. Unlike previous approaches of freezing language models that require large-scale aligned data, our proposed architecture is able to achieve superior performance on small-scale datasets. We conduct extensive experiments to improve the general multimodal capabilities and visual grounding abilities of the MLLM. Our approach remarkably outperforms previous state-of-the-art methods across various vision-language benchmarks without sacrificing performance on NLP tasks. Code and models are available at https://github.com/360CVGroup/Inner-Adaptor-Architecture.

Introduction

Large Language Models (LLMs) have made substantial progress in recent years, largely attributed to the technique of pre-training and instruction tuning. Building upon this foundation, visual instruction tuning has been proposed to evolve LLMs into Multimodal Large Language Models (MLLMs), thereby endowing them with the capability to interpret and comprehend visual signals (Cha et al. 2024). MLLMs (Liu et al. 2024b; Bai et al. 2023; Tong et al. 2024; Chen et al. 2024b; Xuan et al. 2024) prove beneficial in numerous tasks, such as transcribing the text within an image, generating stories and poems based on an image, or converting screenshots of webpages into code (Laurençon et al. 2024). Historically, these tasks have been regarded as challenging for conventional vision-language models. MLLMs exhibit considerable promise in executing these complex, diverse real-world tasks, enabling more natural and human-like interactions (Lu et al. 2024).

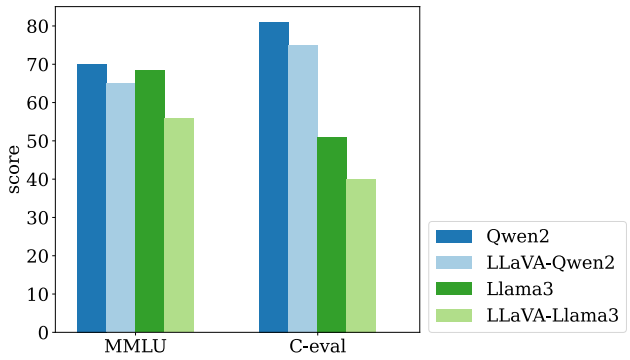


Figure 1: Results before and after training LLaVA-1.5 architecture based on Qwen2 and Llama3 language models on text-only evaluation set MMLU and C-eval.

Typically, the operation of a MLLM begins with feeding an image into a visual encoder, such as CLIP (Radford et al. 2021) or SigLIP (Zhai et al. 2023), to extract a high-dimensional feature representation. This feature is subsequently transformed through a projection layer to align with the dimension of the large language model. The resulting features, often referred to as image tokens, are concatenated with text tokens and fed into the large language model. This process enables the MLLM to generate responses based on user instructions and input images.

In the current common MLLM (Liu et al. 2024a; Bai et al. 2023), when image and text tokens are fed into the large language model, the LLM is typically unfrozen for further training. This strategy has led to significant advancements in the MLLM model. Consequently, it predictably leads to a degradation in the understanding ability of the large language model. To validate this hypothesis, we conduct experiments on the LLaVA-1.5 (Liu et al. 2024a) architecture using the 1.2M-size open-source dataset provided by (Liu

*These authors contributed equally.

†Corresponding author.

et al. 2024a), which contains a limited amount of plain text data, as illustrated in Figure 1. We compare the results before and after training the LLaVA-1.5 architecture, based on the Qwen2 (Yang et al. 2024) and Llama3 (Meta 2024) language models, respectively. The performance of the language model declines significantly on both the MMLU (Hendrycks et al. 2020) and C-Eval (Huang et al. 2023) text-only evaluation sets.

It appears reasonable to posit an explanation for this phenomenon within the field of deep learning. When a model is predominantly trained on a single type of data, it may experience a phenomenon known as catastrophic forgetting. For an MLLM to achieve outstanding image-text comprehension, it is essential to collect a substantial amount of image-text interaction data for training. As observed in Figure 1, training with image-text data results in a decline in language ability. Despite attempts by MLLM such as LLaVA to incorporate some text-only data into their training process, this still leads to a reduction in the model's comprehension.

One direct method to prevent the degradation of LLM performance is to freeze the large language model during the training of MLLM. However, current methods employing this approach (Li et al. 2023a; Zhu et al. 2023) have consistently struggled to achieve powerful multimodal capabilities. To address these challenges, we propose a new training paradigm with an inner-adaptor architecture that significantly enhances multimodal competencies without affecting the original language modeling capabilities. This approach can seamlessly support both multimodal and textual workflows. We evaluate this training paradigm across a spectrum of tasks, including general multimodal capabilities and visual grounding proficiencies. Distinct from previous approaches of freezing language modeling that require large-scale aligned data, our proposed scheme demonstrates effectiveness with a considerably smaller dataset. Comprehensive testing on a suite of benchmarks, including MME, MMBench, MMMU, and RefCOCO, has substantiated the superior performance of our structure. We hope that this approach will provide a reference for future research in open-source MLLM.

Related Work

Large Language Models. The landscape of Natural Language Processing (NLP) has undergone a revolutionary transformation, driven by the advent and continuous refinement of Large Language Models (LLMs). A pivotal moment in this evolution is the first appearance of the transformer architecture, which serves as a key catalyst, giving rise to pioneering language models like BERT (Devlin et al. 2018) and OPT (Zhang et al. 2022). These models showcase an unprecedented level of linguistic comprehension, significantly advancing the state-of-the-art in NLP. A critical breakthrough comes with introducing the Generative Pre-trained Transformer (GPT) series (Brown et al. 2020), which pioneer an auto-regressive language modeling approach, setting a new standard for language prediction and generation capabilities. Subsequent iterations, including Mixtral (Jiang et al. 2024), GPT-4 (Achiam et al. 2023), and Llama3 (Meta 2024), have not only maintained but also amplified this momentum, displaying superior performance on intricate language processing challenges. Moreover, the fusion of LLMs with specialized visual tasks showcases the models' adaptability and broadens their scope, indicating their potential to transcend conventional text-based operations into multimodal interactions. This expansion highlights the transformative role LLMs can assume when incorporated into diverse domains, providing a rich ground for innovation and exploration.

Multimodal Large Language Models. The advancement of Large Language Models (LLMs) has kindled a growing interest in extending their foundational competencies to incorporate the visual domain, thereby giving birth to multimodal Large Language Models (MLLMs). The works on MLLMs (Xie et al. 2023; Li et al. 2023b,a; Bai et al. 2023; Liu et al. 2024b; Laurençon et al. 2024; Chen et al. 2024b) typically follow a tripartite architecture: a visual encoder, a vision-language connector, and a large language model. Notably, BLIP-2 (Li et al. 2023a) and Flamingo (Alayrac et al. 2022) introduce the Q-Former/Resampler as a bridge between vision and language, whereas LLaVA (Liu et al. 2024b) and MiniGPT4 (Zhu et al. 2023) refine this connection via a linear layer. Cambrian-1 (Tong et al. 2024) proposes a dynamically adaptive connector that integrates high-resolution visual features with LLMs while reducing the number of tokens. To enhance their multimodal performance, contemporary MLLMs mainly fine-tune the LLM and connector using visual instruction tuning data. These models leverage meticulously curated instruction datasets, showcasing an effective strategy that highlights their robust capabilities. However, a common oversight lies in the maintenance of language abilities. Long term multimodal training often leads to degradation of language proficiency. CogVLM (Wang et al. 2023) seeks to address this by integrating a trainable visual expert into the language model, but still trains the LLM during supervised fine-tuning, resulting in a degradation of language capability. DeekSeek-VL (Lu et al. 2024) maintains a 70% proportion of language data to preserve the integrity of language knowledge within the model, but incurs a considerable training cost. Departing from these conventional training paradigms of MLLMs, we introduce the inner-adaptor architecture. This design is specifically tailored to preserve the NLP performance of the MLLM while facilitating a seamless augmentation of its multimodal capabilities.

Methodology

Overview. As illustrated in Figure 2, our approach enables the simultaneous execution of two high-quality workflows post-deployment: one for multimodal interactions and the other for text-only conversations. Both workflows leverage the transformer layers of the large language model. The multimodal interaction workflow encompasses: (1) an image encoder and a projector, utilized for extracting high-quality image features and achieving vision-language alignment, respectively, (2) the transformer layers of the large language model, which remain frozen during training, and (3) the

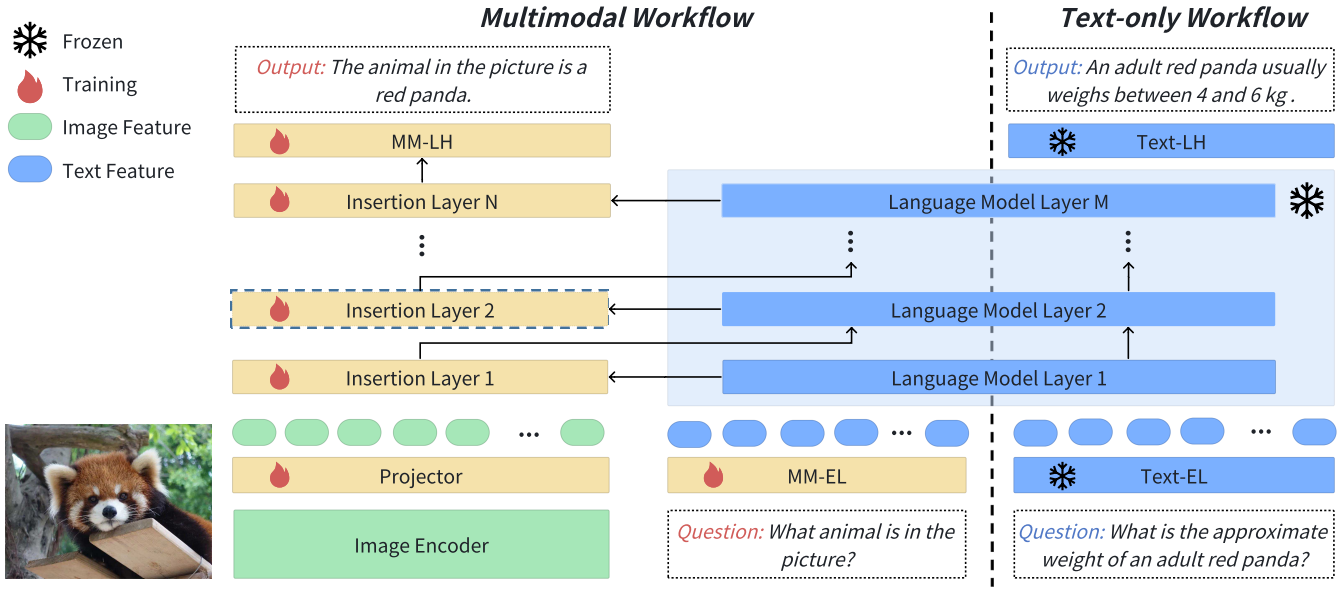


Figure 2: Overview of the proposed architecture, which mainly consists of two workflows: the Multimodal Workflow and the Text-only Workflow. The multimodal workflow, beyond the necessary image encoder and projector, integrates the Inner-Adaptor Architecture, including insertion layers, an embedding layer, and a language model head. Both workflows share the same large language model. The number of insertion layers is variable, where $N \leq M$. In this context, *MM* denotes MultiModal, *EL* stands for Embedding Layer, and *LH* represents the Language model Head.

inner-adaptor architecture, which comprises insertion layers, an embedding layer, and a language model head specifically designed for multimodal inputs. Conversely, the text-only conversation workflow solely employs the constituent elements of the original language model, without resorting to the specialized multimodal components.

Image Encoder and Projector. Following LLava-1.5 (Liu et al. 2024a), we utilize the CLIP ViT-L/14 (Radford et al. 2021) image encoder with an input resolution of 336px. Subsequently, we employ a vision-language projector composed of a two-layer MLP to integrate the vision features with LLMs.

Large Language Model. We employ the Llama3-8B (Meta 2024) as the base language model throughout the training process.

Inner-Adaptor Architecture. To achieve multimodal comprehension, it is essential to integrate trainable parameters into MLLMs. LLaVA (Liu et al. 2024b) makes the projector and the large language model trainable during visual instruction tuning, but leads to the performance degradation on NLP tasks. Flamingo (Alayrac et al. 2022) employs cross-attention with a gating mechanism to introduce image information into the model, facilitating a deep fusion of original image features with text features prior to each layer of the language model. However, this approach requires a considerable volume of pre-training data to train effective cross-attention layers and gating values, which can be computationally costly. Furthermore, the final performance of the model falls short of expectations.

Drawing insights from recent works (Chen et al. 2024a; Tong et al. 2024), we recognize that the self-attention layer can assimilate image features as prior prompts, thus eliminating the necessity of cross-attention for the obligatory incorporation of image features. In alignment with this perspective, we embark on exploratory research. Referencing Figure 3(a), we are inspired by the prevalent ControlNet (Zhang, Rao, and Agrawala 2023) architecture. The operation of a specific layer can be succinctly expressed as follows:

$$X_{out} = \phi_{fl}(X_{in}) + G(\phi_{il}(X_{in})), \tag{1}$$

where ϕ_{fl} and ϕ_{il} denote the frozen language model (LM) layer and the insertion layer, respectively. Here, X_{in} represents the multimodal input, X_{out} denotes the multimodal output, and G indicates a gating layer initialized at zero. The insertion layer is a transformer decoder layer, comprising the self-attention layer, layer normalization, feed forward network, etc. It is consistent with the parameter scale of a transformer layer in the large language model. For instance, if we target the $22th$ layer, the initial parameters of the corresponding insertion layer are derived from the $22th$ language model layer. Nonetheless, the ControlNet-based design did not yield satisfactory performance.

Referring to Figure 3(b), we endeavor to refine the ControlNet structure. Specifically, we eliminate the feature propagation between insertion layers. Instead, the output of the LM layer serves as the input to the insertion layer. Our expectation is that each frozen LM layer will accommodate multimodal data through a distinct insertion layer and gating layer, with the insertion layer no longer being directly

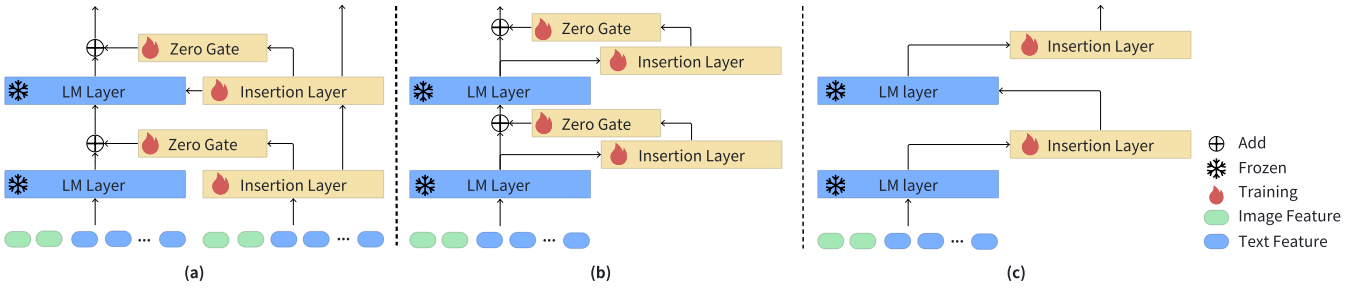


Figure 3: Structural exploration of the Inner-Adaptor Architecture. Figure (a) is a architecture inspired by the ControlNet design; Figure (b) is an improvement on Figure (a), mainly canceling the feature propagation between adaptors; Figure (c) is the final scheme.

influenced by subsequent layers. Compared to the design in Figure 3(a), the refined architecture shows significant improvements.

Moreover, we hypothesize that the gating layer may not reach an optimal state through a single round of data training. Consequently, we propose a more streamlined solution, as illustrated in Figure 3(c). The operation of a specific layer within the model can be represented as follows:

$$X_{out} = \phi_{il}(\phi_{fl}(X_{in})). \tag{2}$$

Similar to Scheme (a), if an insertion layer is placed after the $22th$ LM layer, it is initialized from the parameters of the $22th$ frozen LM layer. The number of insertion layers is adjustable.

Additionally, for multimodal training, we introduce a new embedding layer EL_{mm} and a new LM head LH_{mm}, initialized from the original language model's embedding layer EL_{text} and LM head LH_{text}. Throughout all stages of multimodal training, EL_{text} and LH_{text} will remain frozen, while the newly created components will be trained with multimodal data. The experimental results presented in Table 5 validate the effectiveness of this strategy.

We thoroughly explore the distinctions among these architectures and strategies in the ablation study. Ultimately, we select the structure depicted in Figure 3(c), which we designate as the Inner-Adaptor Architecture (IAA).

Configurations	Satge1-PT	Satge2-PT
Trainable modules	Projector	Projector, Inner-adaptor
Learning rate	1e-3	2e-5
Batch size	256	
LR schedule	Cosine decay	
Training steps	2.5K	
Zero-Stage	Zero2	
Warmup ratio	0.03	
Weight decay	0.0	
Optimizer	AdamW	
Optimizer HPs	$\beta_1 = 0.9, \beta_2 = 0.98, \epsilon = 1e-6$	
Configurations	**Instruction-FT**	**Grounding-FT**
Trainable modules	Projector, Inner-adaptor	
Learning rate	2e-5	
Batch size	128	
LR schedule	Cosine decay	
Training steps	6.6K	18K
Zero-Stage	Zero3	
Warmup ratio	0.03	
Weight decay	0.0	
Optimizer	AdamW	
Optimizer HPs	$\beta_1 = 0.9, \beta_2 = 0.98, \epsilon = 1e-6$	

Table 1: The hyperparameters utilized during the training phase are delineated as follows: "-PT" designates the pre-training phase, "-FT" denotes the fine-tuning phase, and "HP" and "LR" signify the hyperparameter and learning rate, respectively.

Experiments

In this section, we first describe the training paradigm of our method with the data utilized in the diverse processes. Subsequently, we conduct evaluation on the general multimodal and visual grounding benchmarks to comprehensively assess our models' visual understanding ability. Finally, we detail the ablation experiments of our method.

Training Paradigm

Pre-training. During the training process of MLLM, the primary objective of the pre-training phase is to enable MLLM to learn the alignment between visual cues and textual descriptions. This stage, also known as the image-text alignment phase, establishes connections between the vision encoder and LLM. In our architectural design, the image encoder and LLM remain frozen throughout all training phases to preserve the inherent foundational knowledge in both vision and language models. The projector and inner-adapter architecture require training to enhance multimodal capabilities. Our empirical investigations reveal that for the inner-adaptor architecture, applying a high learning rate can lead to overflow in training loss. To alleviate this issue, we devise a dual-stage pre-training procedure.

In the first pre-training stage, the model configuration consists of only three components: the image encoder, the projector, and the large language model. The parameters of the image encoder and the large language model are frozen, while a high learning rate of 0.001 is utilized to train a high-quality projector.

In the second pre-training stage, the model architecture

Method	Vision Encoder	Language Model	Data Scale	MME^{P}	$MMB\text{-}EN^{T}$	$MMB\text{-}CN^{T}$	$MMMU^{v}$
Training with the LLM unfrozen							
mPLUG-Owl(Ye et al. 2023)	CLIP-ViT-L	Llama2 (7B)	1.1B	967.3	49.4	-	-
Qwen-VL-Chat (Bai et al. 2023)	CLIP-ViT-G	Qwen (7B)	1.5B	1487.6	61.8	56.3	37
CogVLM (Wang et al. 2023)	EVA2-CLIP-ViT-E	Vicuna-v1.5 (7B)	1.5B	1439.7	65.8	55.9	37.3
mPLUG-Owl2 (Ye et al. 2024)	CLIP-ViT-L	Llama2 (7B)	400M	1450.2	66.0	60.3	34.7
LLaVA-1.5(Liu et al. 2024b)	CLIP-ViT-L	Vicuna-v1.5 (7B)	1.2M	1510.7	66.5	59.0	35.7
LLaVA-1.5(Liu et al. 2024b)	CLIP-ViT-L	Vicuna-v1.5 (13B)	1.2M	1531.3	69.2	65.0	37.0
Honeybee (Cha et al. 2024)	CLIP-ViT-L	Vicuna-v1.5 (7B)	208M	1584.2	70.1	-	-
Yi-VL (AI et al. 2024)	CLIP-ViT-H	Yi (6B)	125M	-	68.4	66.6	39.1
DeepSeek-VL (Lu et al. 2024)	SAM-B and SigLIP-L	DeepSeek (7B)	103M	-	73.8	**71.4**	36.6
LLaVA-Llama3 (Contributors 2024)	CLIP-ViT-L	Llama3 (8B)	1.2M	1506.0	68.9	61.6	36.8
Training with the LLM frozen							
OpenFlamingov2 (Awadalla et al. 2023)	CLIP-ViT-L	MPT (7B)	3B	-	5.7	14.4	28.8
Llama-AdapterV2(Gao et al. 2023)	CLIP-ViT-L	Llama2 (7B)	0.6M	972.7	41.0	-	-
MiniGPT-4 (Zhu et al. 2023)	EVA-CLIP-ViT-G	Vicuna (13B)	5.1M	866.6	-	-	-
BLIP-2 (Li et al. 2023a)	EVA-CLIP-ViT-G	FlanT5XXL	129M	1293.8	-	-	-
InstructBLIP (Dai et al. 2023)	EVA-CLIP-ViT-G	Vicuna (13B)	130M	1212.8	44.0	-	-
IAA-$8^{\dagger}$	CLIP-ViT-L	Llama3 (8B)	1.2M	1560.2	69.9	64.2	39.0
IAA-8	CLIP-ViT-L	Llama3 (8B)	1.5M	1581.8	72.7	69.2	39.8
IAA-14	CLIP-ViT-L	Llama3 (8B)	1.5M	**1591.5**	**74.9**	70.5	**39.9**

Table 2: Results on general multimodal benchmarks, where the data scale of 1.2M uniformly represents the data provided by LLaVA (Liu et al. 2024b). IAA-$8^{\dagger}$ represents the model trained using 1.2M data.

Method	MMLU↑	C-Eval ↑	BBH↑	Humaneval ↑	Math ↑
LLaVA-Llama3	55.8	40.5	44.6	38.4	12.3
IAA-$8^{\dagger}$	68.4	51.3	52.8	59.2	27.8

Table 3: Comparison on Text-only Benchmarks. IAA-$8^{\dagger}$ denotes the model trained using the same 1.2M data as LLaVA-Llama3. IAA-$8^{\dagger}$ is not impaired in terms of NLP ability, but LLaVA-Llama3 presents deteriorated results.

is expanded to incorporate the inner-adaptor for multimodal tasks. The training parameters now include both the projector and the newly integrated structures. The projector is initialized with the parameters derived from the preceding stage. For this stage, a lower learning rate of 2e-5 is adopted.

Throughout the pre-training stages, the dataset employed consists of 558k image-text aligned pairs sourced from (Liu et al. 2024b) and an additional 100K pairs from (Chen et al. 2024a). (Chen et al. 2024a) provides a total of 664K image-text aligned data. We translate the first 100k pairs into Chinese and incorporated them into the training process to fortify the model's understanding of Chinese tasks. Over the course of these stages, we utilize a cumulative total of 658K data pairs.

Instruction Fine-tuning. We perform instruction fine-tuning based on the model obtained from the second pre-training stage. Throughout this stage, the parameters of the large language model and the image encoder remain frozen. The dataset includes the fine-tuning dataset of 665K samples proposed by (Liu et al. 2024b), along with additional datasets including DocVQA (50K) (Mathew, Karatzas, and Jawahar 2021), VSR (10K) (Liu, Emerson, and Collier 2023), ScienceQA (21K) (Lu et al. 2022), and an in-house dataset (78.5K). Similar to the pre-training stage, we translate the first 40K entries of the 664K fine-tuning data proposed by (Chen et al. 2024a) into Chinese and incorporate them into the instruction fine-tuning dataset. The aggregate quantity of data utilized in this stage amounts to 865K.

Grounding Fine-tuning. Building upon the model fine-tuned with instructions, we further train a model specialized in visual grounding. The data used in this stage comprises RefCOCO (Kazemzadeh et al. 2014), COCO (Lin et al. 2014), Flickr30k Entities (Plummer et al. 2015), Objects365 (Shao et al. 2019), aggregating to approximately 2M data instances. These datasets improves the model's capability of localizing fine-grained visual details. The inclusion of COCO and Objects365 assists the model in improving its ability to localize multiple targets.

Implementation details. The detailed training information is summarized in Table 1, mainly covering the hyper-parameters used during the four-stage training process. The entire four-stage can be executed on a single node A800×8 in 48 hours. All experiments utilize the zero technology provided by (Rajbhandari et al. 2020) and the flash-attention v2 provided by (Dao 2023).

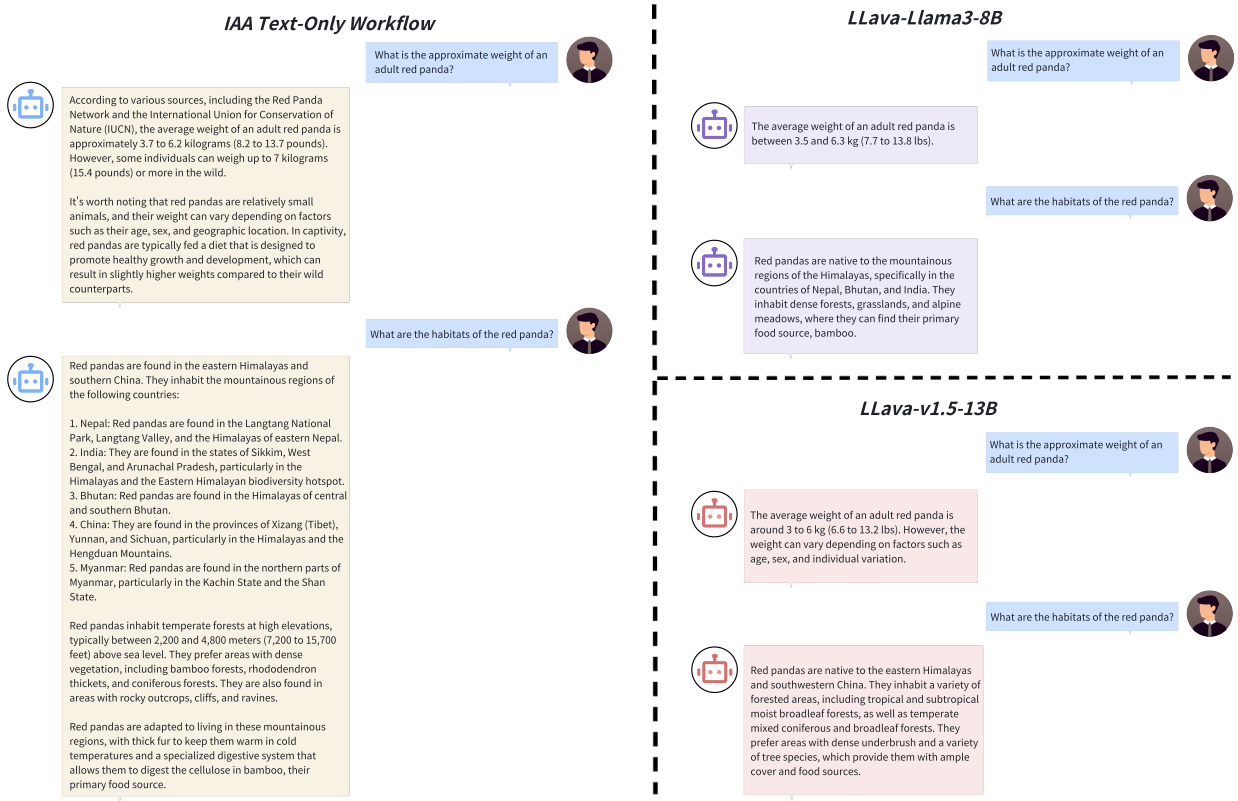


Figure 4: Comparison on text-only question answering.

Experimental Results

Main Results on General Multimodal Benchmarks. To assess the multimodal capabilities of our approach, we employ widely recognized benchmarks that are closely related to multimodal tasks: MME^{P} (Fu et al. 2023), $\text{MMBench-EN}^{\text{T}}$ (Liu et al. 2023), $\text{MMBench-CN}^{\text{T}}$ (Liu et al. 2023), and MMMU^{v} (Yue et al. 2024). These benchmarks are renowned for presenting significant challenges across a diverse range of practical tasks. For evaluation purposes, we adhere to a zero-shot testing protocol, a strict methodology that tests models on unseen data without additional training. Moreover, we categorize comparative methods into two distinct categories: those trained with a frozen language model and those trained with an unfrozen language model. To provide a comprehensive analysis, we show the scale of the data utilized for each method, along with the variations in the image encoders employed. Detailed results of our evaluations are tabulated in Table 2. To ensure a fair and equitable comparison, we choose methods that leverage a base language model with a comparable parameter scale, and the reported metrics for competing methods are based solely on officially published data, avoiding any local testing results.

Owing to the inherent strengths of our proposed architecture, our method exhibits substantial superiority over those trained with frozen language model. As the current mainstream approach, models trained with unfrozen language models typically achieve better multimodal performance, albeit at the cost of diminished NLP capabilities. We list several state-of-the-art methods adhering to this training paradigm. Compared to Honeybee (Cha et al. 2024), Yi-VL (AI et al. 2024), and Deepseek-VL (Lu et al. 2024), our method achieves competitive or even superior performance on certain metrics, with an extremely small training data scale. Using the same data scale of 1.2 million, IAA-8 outperforms LLaVA-Llama3. Additionally, IAA-14 with 14 insertion layers achieves better results than IAA-8 with an 8-layer configuration. Furthermore, we compare our approach with LLaVA-Llama3 (Contributors 2024) on NLP benchmarks, including MMLU and C-Eval. The results of NLP benchmarks are summarized in Table 3. Our language model is not impaired in terms of NLP ability, but LLaVA-Llama3 trained on the same data shows deteriorated results on both MMLU and C-Eval. Our method surpasses LLaVA-Llama3 across all metrics, indicating that our architecture is superior to the mainstream LLaVA architecture. The performance of various models on the plain text dialog task is illustrated in Figure 3. It is evident that the text-only workflow of the Inner-Adaptor Architecture (IAA) preserves the original conversational capabilities of the language model. In contrast, open-source multimodal large language models such as LLaVA-Llama3 and LLaVA-v1.5 are more impacted by multimodal data. When queried with the same question, LLaVA-Llama3 and LLaVA-v1.5 produce notably shorter responses. This is directly related to the fact that a

Method	Grounding Data Scale	RefCOCO			RefCOCO+			RefCOCOg	
		val	testA	testB	val	testA	testB	val	test
KOSMOS-2 (Peng et al. 2023)	20M	52.3	57.4	47.3	45.5	50.7	42.2	60.6	61.7
OFA-L (Wang et al. 2022)	10M	80.0	83.7	76.4	68.3	76.0	61.8	67.6	67.6
Shikra (Chen et al. 2023b)	4M	87.0	90.6	80.2	81.6	87.4	72.1	82.3	82.2
MiniGPT-v2 (Chen et al. 2023a)	∼21M	88.7	91.7	85.3	80.0	85.1	74.5	84.4	84.7
Ferret (You et al. 2023)	8.7M	87.5	91.4	82.5	80.1	87.4	73.1	83.9	84.8
PINK (Xuan et al. 2024)	5M	88.7	92.1	84.0	81.8	88.2	73.9	83.9	84.3
IAA-8	2M	89.2	92.6	83.7	82.1	88.6	73.7	84.4	84.7
IAA-14	2M	**90.2**	**92.9**	**85.4**	**83.4**	**89.0**	**76.7**	**85.0**	**85.1**

Table 4: Comparisons on visual grounding benchmarks. Our approach achieves competitive performance trained on relatively limited datasets.

large amount of the multimodal training data has shorter text lengths. Fine-tuning the large language model affects its ability to fully understand content and generate more comprehensive responses.

Results on Visual Grounding Benchmarks. To evaluate the effectiveness of our model in the visual grounding task, we perform evaluations utilizing the widely accepted benchmarks RefCOCO (Kazemzadeh et al. 2014), RefCOCO+ (Yu et al. 2016), and RefCOCOg (Mao et al. 2016), with the corresponding results illustrated in Table 4. The methods for comparison are all models trained for the grounding task under an auto-regressive strategy. The results reveal that our method is capable of achieving competitive performance, even when trained on relatively limited datasets. In our analysis, to ensure fairness, we exclude models trained on extremely large-scale datasets, such as CogVLM-grounding (Wang et al. 2023) with 1.5B image-text pairs and 40M grounding data, as well as those leveraging pre-trained object detection models, exemplified by LLaVA-Grounding (Zhang et al. 2023) and Groma (Ma et al. 2024).

Efficiency in Deployment. Currently, high-performance multimodal models typically require the unfreezing of the large language model for training. CogVLM (Wang et al. 2023) highlights the substantial difficulty in developing a model that excels in both multimodal comprehension and visual grounding tasks simultaneously. To address this, it adopts a dual-model strategy, specifically training one model for general multimodal capabilities and another for visual grounding abilities. In this context, deploying a high-quality language model, a multimodal model with outstanding general performance, and a model endowed with proficient visual grounding skills concurrently on a single GPU would demand an estimated 50GB of memory. Our proposed approach, facilitated by the inner-adaptor architecture, ingeniously combines superior general multimodal competencies and robust visual grounding capacities, while concurrently safeguarding the inherent prowess of the original large language model. Specifically, with an 8-layer inner-adaptor configuration, our model exhibits a significantly reduced memory footprint, hovering around 30GB.

Ablation Study

Structure Analysis. In the exploration of the structure, we furnish quantitative results for validation in Table 5. With an 8-layer insertion scheme as our baseline configuration, we observe that incremental architectural enhancements consistently improve performance metrics across the board. Specifically, the comparison between rows 1, 2, and 4 highlights the benefits of architectural refinement. Moreover, the contrast between rows 3 and 4 demonstrates that the integration of a specialized embedding layer and language model head for multimodal data processing significantly boosts performance.

Comparison of Training Stages. Through empirical evidence detailed in Table 6, we validate the effectiveness of our two-stage pre-training methodology. It can be observed that the model lacking the first stage of alignment training exhibits notably poorer performance. When the projector and insertion layers are engaged in joint pre-training, it is essential to maintain a learning rate of approximately 2e-5 to prevent loss overflow. However, this strategy leads to suboptimal alignment training for the projector, which negatively affects the model's final performance. Furthermore, although the model performs adequately when skipping the second pre-training stage, it ultimately fails to replicate the outstanding results achievable through the complete two-stage pre-training process. This disparity emphasizes the critical significance of the additional pre-training stage in enhancing the model's overall effectiveness.

Impact of Insertion Layer Quantities. We explore the effect of varying numbers of insertion layers, which are presented in Table 7. The experimental results indicate that increasing the number of insertion layers from 8 to 14 yields enhancements in all performance metrics. However, it is imperative to acknowledge that an increase in insertion layers simultaneously impacts the model's efficiency. We advocate that an 8-layer configuration is adequate to effectively address foundational requirements.

Model architecture	Trainable modules					MME^P	$MMB\text{-}EN^T$	$MMB\text{-}CN^T$	$MMMU^v$
	Projector	I-Layers(8)	EL_{mm}	LH_{mm}	Zero-Gates				
Figure 3(a)	✓	✓	✓	✓	✓	1425.4	72.4	65.0	38.2
Figure 3(b)	✓	✓	✓	✓	✓	1556.0	72.7	68.5	39.6
Figure 3(c)	✓	✓	×	×	×	1563.4	72.6	68.7	39.6
Figure 3(c)	✓	✓	✓	✓	×	1581.8	72.7	69.2	39.8

Table 5: Ablation study for the exploration of inner-adaptor related structures.

Training Stages			MME^P	$MMMU^v$
Satge1-P	Satge2-P	Instruction-F		
×	✓	✓	1512.1	39.3
✓	×	✓	1565.4	39.5
✓	✓	✓	1581.8	39.8

Table 6: Comparison of the training stages.

Number of I-Layers	MME^P	$MMB\text{-}EN^T$	$MMB\text{-}CN^T$
8	1581.8	72.7	69.2
14	1591.5	74.9	70.5
22	1531.7	76.0	70.4

Table 7: Ablations on the number of insertion layers.

Training Data Influence Assessment. To delineate the impact of data on model performance, we present comparative results in Table 8. The baseline, outlined in the first row, showcases the performance of LLaVA-Llama3 (Contributors 2024) utilizing the LLaVA architecture and the 1.2 million dataset provided by (Liu et al. 2024b). Subsequent experimentation, as delineated in the second row, emphasizes the pronounced superiority of our proposed architecture over LLaVA. Additionally, we enrich the training corpus with an extra 0.3 million records, mainly encompassing Chinese data. As a result, our model achieves substantial improvements in all metrics, especially on the Chinese evaluation set MMBench-CN^T.

Limitations The method of extending multimodal capabilities by freezing the language model will introduce certain additional parameters. Compared to the approach of training with an unfrozen language model, the inference speed of the model will be reduced. To mitigate this issue, we extend the key-value cache mechanism to the insertion layers. Based on the MME dataset, compared to the LLaVA architecture, the average inference time of our 8-layer structure increases from 0.103s to 0.124s, which we deem to be within a relatively reasonable range.

Conclusion

In this paper, we introduce the Inner-Adaptor Architecture, which is designed to enhance the general multimodal and visual grounding capabilities of LLMs. Through a series of architectural exploration experiments, we demonstrate that training with a frozen language model can surpass the multimodal performance of the models with fine-tuned LLMs. Our proposed model has achieved state-of-the-art performance across a multitude of publicly available evaluation datasets. Moreover, after deployment, our approach incorporates dual workflows, thereby preserving the NLP proficiency of the language model. The flexibility of the Inner-Adaptor Architecture provides the potential for extension to additional modalities, which is a direction for future exploration.

	MME^P	$MMB\text{-}EN^T$	$MMB\text{-}CN^T$
LLaVA-Llama3 (1.2M)	1506.0	68.9	61.6
IAA-8 (1.2M)	1560.2	69.9	64.2
IAA-8 (1.5M)	1581.8	72.7	69.2

Table 8: The impact of the training data.

References

Achiam, J.; Adler, S.; Agarwal, S.; Ahmad, L.; Akkaya, I.; Aleman, F. L.; Almeida, D.; Altenschmidt, J.; Altman, S.; Anadkat, S.; et al. 2023. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*.

AI, .; :; Young, A.; Chen, B.; Li, C.; Huang, C.; Zhang, G.; Zhang, G.; Li, H.; Zhu, J.; Chen, J.; Chang, J.; Yu, K.; Liu, P.; Liu, Q.; Yue, S.; Yang, S.; Yang, S.; Yu, T.; Xie, W.; Huang, W.; Hu, X.; Ren, X.; Niu, X.; Nie, P.; Xu, Y.; Liu, Y.; Wang, Y.; Cai, Y.; Gu, Z.; Liu, Z.; and Dai, Z. 2024. Yi: Open Foundation Models by 01.AI. arXiv:2403.04652.

Alayrac, J.-B.; Donahue, J.; Luc, P.; Miech, A.; Barr, I.; Hasson, Y.; Lenc, K.; Mensch, A.; Millican, K.; Reynolds, M.; et al. 2022. Flamingo: a visual language model for few-shot learning. In *Advances in neural information processing systems*, volume 35, 23716–23736.

Awadalla, A.; Gao, I.; Gardner, J.; Hessel, J.; Hanafy, Y.; Zhu, W.; Marathe, K.; Bitton, Y.; Gadre, S.; Sagawa, S.; Jitsev, J.; Kornblith, S.; Koh, P. W.; Ilharco, G.; Wortsman, M.; and Schmidt, L. 2023. OpenFlamingo: An Open-Source Framework for Training Large Autoregressive Vision-Language Models. *arXiv preprint arXiv:2308.01390*.

Bai, J.; Bai, S.; Yang, S.; Wang, S.; Tan, S.; Wang, P.; Lin, J.; Zhou, C.; and Zhou, J. 2023. Qwen-vl: A frontier large vision-language model with versatile abilities. *arXiv preprint arXiv:2308.12966*.

Brown, T.; Mann, B.; Ryder, N.; Subbiah, M.; Kaplan, J. D.; Dhariwal, P.; Neelakantan, A.; Shyam, P.; Sastry, G.; Askell, A.; et al. 2020. Language models are few-shot learners.

Learning Global Controller in Latent Space for Parameter-Efficient Fine-Tuning

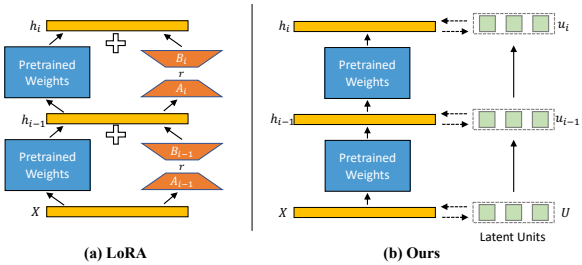
Zeqi Tan[1], **Yongliang Shen**[1], **Xiaoxia Cheng**[1], **Chang Zong**[1], **Wenqi Zhang**[1], **Jian Shao**[1], **Weiming Lu**[1,2]*, **Yueting Zhuang**[1]*
[1]School of Computer Science and Technology, Zhejiang University
[2]Alibaba-Zhejiang University Joint Research Institute of Frontier Technologies
{zqtan,syl,luwm,yzhuang}@zju.edu.cn

Abstract

While large language models (LLMs) have showcased remarkable prowess in various natural language processing tasks, their training costs are exorbitant. Consequently, a plethora of parameter-efficient fine-tuning methods have emerged to tailor large models for downstream tasks, including low-rank training. Recent approaches either amalgamate existing fine-tuning methods or dynamically adjust rank allocation. Nonetheless, these methods continue to grapple with issues like local optimization, inability to train with full rank and lack of focus on specific tasks. In this paper, we introduce an innovative parameter-efficient method for exploring optimal solutions within latent space. More specifically, we introduce a set of latent units designed to iteratively extract input representations from LLMs, continuously refining informative features that enhance downstream task performance. Due to the small and independent nature of the latent units in relation to input size, this significantly reduces training memory requirements. Additionally, we employ an asymmetric attention mechanism to facilitate bidirectional interaction between latent units and freezed LLM representations, thereby mitigating issues associated with non-full-rank training. Furthermore, we apply distillation over hidden states during the interaction, which guarantees a trimmed number of trainable parameters. Experimental results demonstrate that our approach achieves state-of-the-art performance on a range of natural language understanding, generation and reasoning tasks.

1 Introduction

Large language models (LLMs) (Brown et al., 2020; Ouyang et al., 2022; Chowdhery et al., 2022; Zhang et al., 2022; Zeng et al., 2023; Touvron et al., 2023a), exemplified by ChatGPT, have garnered substantial attention within the scholarly and industrial realms owing to their remarkable efficacy in a plethora of natural language processing undertakings. However, full training of LLM is time-consuming and labor-intensive. Besides the high training overhead, maintaining a replica for each task introduces significant storage redundancy.



(a) LoRA (b) Ours

Figure 1: The white plus indicates summing the two hidden state values bitwise, and the dashed arrow represents an optional bidirectional exchange of information. (a) LoRA (Hu et al., 2022) approximates the update of each weight matrix with a pair of A_i and B_i, involving various modules in transformer layers. (b) Instead of this idea of local approximation, our approach treats LLM as a feature extractor and uses a set of latent units to iteratively perform the exchange of information with LLM, further exploiting LLM's power.

To address these issues, researchers either add extra neural modules (Rebuffi et al., 2017; Houlsby et al., 2019; Pfeiffer et al., 2020) or model incremental updates (Zaken et al., 2021; Guo et al., 2021; Hu et al., 2022). More recently, researchers either merge existing fine-tuning methods (He et al., 2022a; Wang et al., 2023a) or dynamically adjust rank allocation (Zhang et al., 2023; Ding et al., 2023a). He et al. (2022a) presents a unified framework and enables an efficient combination of existing fine-tuning methods. Ding et al. (2023a) dynamically allocates the parameter budget among weight matrices. However, these methods still have some flaws. **First**, local optimization is a notorious problem. As shown in Figure 1 (a), LoRA (Hu et al., 2022) approximates the update of each weight matrix with a pair of A_i and B_i. These

* Corresponding author

Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 4044–4055

low rank-based methods use numerous pairs of low rank matrices to fit local parameters and neglect global control, resulting in suboptimal performance. Zhang et al. (2023); Ding et al. (2023a) remains with this problem even though it makes the unalterable rank in LoRA to be adaptive. **Second**, these methods mostly suffer from inability to train with full rank. He et al. (2022a) finds that this problem manifests itself even more severely in FFNs, and therefore assigns more parameters to FFNs than to attention modules to alleviate this problem. However, as in previous approaches (Pfeiffer et al., 2020; Guo et al., 2021), they still takes a bitwise summing approach to varying the output probabilities, limiting the expressive power of the model. **Third**, these methods focus only on approximate updates to LLMs and do not adequately consider task-specific relevant features (Wang et al., 2023b). Modeling the topics or labels of specific tasks will result in performance gains.

To address these issues, we propose to learn a global controller (GloC) for parameter-efficient training in latent space, in which we use a set of latent units to iteratively distill information features from LLM. As shown in Figure 1 (b), we treats LLM as a feature extractor and uses a set of latent units to perform the exchange of information with LLM. By the nature of the small and independent nature of the latent units relative to the size of the input, this greatly reduces the requirement for training memory. We consider this set of latent units as a global controller that runs through all layers of the large language model, fully exploiting the capabilities of LLM. In addition, we employ asymmetric attentional mechanisms to facilitate bidirectional interactions between latent units and freezed representations, hence mitigating the problems associated with non-full-rank training. Further, we apply a distillation technique to the hidden states during the interaction to compress the hidden state size to a very small scale, which ensures fewer trainable parameters. We also find in our experiments that these latent units learn task-specific relevant features that show strong statistical correlation with task labels. Our main contributions are as follows:

- We consider parameter efficient fine-tuning from a novel perspective that learns a global controller to interact with LLMs in an informative manner. Based on a small set of latent units, we steer the large language model from a global angle, seeking optimal performance.
- We design the asymmetric attention mechanism and distillation compression module during the information exchange to reduce the training memory while mitigating the problem of non-full-rank training.
- Extensive experiments on a range of natural language understanding, generation, and reasoning tasks show that our model reaches the state-of-the-art and significantly outperforms a series of robust baselines. The experimental results also indicate that these latent units model task-specific features.

2 Related Work

2.1 Parameter-Efficient Fine-Tuning

Parameter-efficient fine-tuning (PEFT) is a set of methods that optimize only a small fraction of the parameters, keeping the backbone model frozen to adapt to downstream subtasks. Mainstream approaches either add external neural modules (Houlsby et al., 2019; Li and Liang, 2021; Lester et al., 2021) or model incremental updates (Zaken et al., 2021; Guo et al., 2021; Hu et al., 2022). Specifically, the methods for adding extra modules include Adapter (Houlsby et al., 2019; Rebuffi et al., 2017; Pfeiffer et al., 2020), Prefix (Li and Liang, 2021) and Prompt Tuning (Lester et al., 2021). Adapter inserts small neural modules called adapters between layers of the backbone model, while Prefix and Prompt Tuning appends additional trainable prefix tokens to the input or hidden layers, similar work also includes P-tuning (Liu et al., 2021). Another mainstream of approaches model incremental updates of pre-training weights without modifying the model structure. (Zaken et al., 2021) only fine-tunes bias vectors in the backbone model, and diff-pruning (Guo et al., 2021) learns a sparse parameter update vector. LoRA (Hu et al., 2022) approximates the update of each weight matrix with a pair of low-rank matrices.

Recent work either customize existing fine-tuning methods (He et al., 2022a; Wang et al., 2023a) or dynamically adjust rank allocation (Zhang et al., 2023; Ding et al., 2023a). He et al. (2022a) presents an efficient combination of existing fine-tuning methods, and Wang et al. (2023a) utilizes low-rank techniques to highly parameterize skills in the multi-task. Zhang et al. (2023); Ding et al. (2023a) both dynamically allocates the parameter budget among weight matrices. Zhang et al.

(2023) prunes the singular values of unimportant updates, while Ding et al. (2023a) use a gate unit to controll the cardinality of rank. However, they all suffer from the problem of local optimization and the inability to train with full rank.

2.2 Controller View for PEFT

Yang and Liu (2022) proposes to explain prefix tuning from a controller perspective. Ding et al. (2023b) extends the controller perspective to a broader set of PEFT approaches. They argue that the essence of PEFT lies in the regularized layered hidden state transformation process. The proposed global controller is inspired by Lee et al. (2019); Jaegle et al. (2021), which are designed to address high-dimensional multimodal inputs. Unlike their attempts to compress inputs of tens of thousands of dimensions (e.g., pixels) into lower units for probabilistic generation, our approach establishes bi-directional channels for information exchange and distilling hidden states.

3 Method

3.1 Preliminaries

Transformer-based Models A typical transformer model is composed of a stack of L transformer (Vaswani et al., 2017) layers, and the modeling process mainly involves the multi-head attention mechanism. For simplicity, we denote the attention as:

$$\text{Attention}\left(Q, K, V\right) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V, \tag{1}$$

where Q, K, V are the query, key and value matrix respectively, and the $1/\sqrt{d_k}$ is the scaling factor. Given the input token embeddings $X \in \mathbb{R}^{n\times d}$, n is length of input token sequence and d is hidden size, the multi-head self-attention (MHA) computes the output on N_h head and concatenates them:

$$\begin{aligned}\text{MHA}(X) &= \text{Concat}\left(\text{head}_1, \cdots, \text{head}_\text{h}\right)\boldsymbol{W}_o, \\ \text{head}_\text{i} &= \text{Attn}\left(E\boldsymbol{W}_q^{(i)}, E\boldsymbol{W}_k^{(i)}, E\boldsymbol{W}_v^{(i)}\right),\end{aligned} \tag{2}$$

where $\boldsymbol{W}_o \in \mathbb{R}^{d\times d}$, $\boldsymbol{W}_q^{(i)}, \boldsymbol{W}_k^{(i)}, \boldsymbol{W}_v^{(i)} \in \mathbb{R}^{d\times d_h}$, d_h is typically set to d/N_h.

PEFT with Transformer In order to provide a comprehensive understanding of the differences between our approach and the previous series of approaches, we do a careful recap here. As in Figure 2, the different peft methods are embedded in

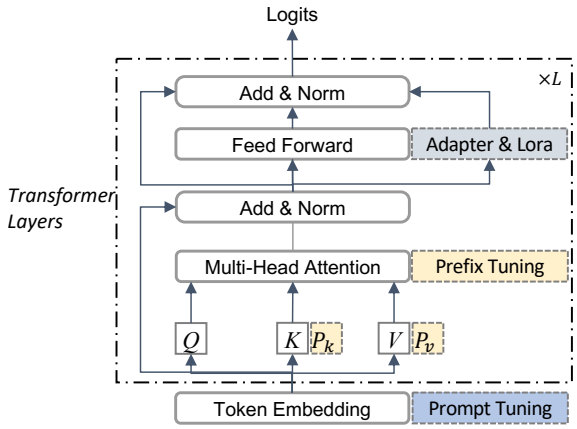


Figure 2: The location of various peft methods in a transformer layer. For simplicity, we only show LoRA approximation to FFN matrices, which can be extended to arbitrary matrices. Similarly, we illustrate parallel adapters, while some earlier adapters are sequential.

different modules of the transformer. It is worth noting that LoRA (Hu et al., 2022) can be used to approximate arbitrary matrices, including the matrices in attention and FFN, although of course approximation of more matrices will result in more parametric quantities. Prefix (Li and Liang, 2021) and Prompt Tuning (Lester et al., 2021) appends additional trainable prefix tokens to the input or hidden attention layers, similar work also includes P-tuning (Liu et al., 2021). Adapter inserts small neural modules called adapters between layers of the backbone model. Houlsby et al. (2019) places two adapters sequentially within one layer of the transformer, one after the multi-head attention and one after the FFN sub-layer, while He et al. (2022a) incorporate extra adapter modules in parallel as in Figure 2. Unlike these approaches that add additional small modules to the submodules of the transformer layers or approximate the update of the local matrices, our proposed approach stands outside of the backbone's transformer layers and delivers the flow of information with a global perspective, as shown on the left side of Figure 3. Moreover, our approach can be summarized as prefix fine-tuning with low-rank attention matrices. We will elaborate on this in the following section.

Latent Units The latent arrays in our approach can be traced back to (Gu et al., 2018; Carion et al., 2020). Carion et al. (2020) refers to them as learnable queries for the set multi-objective detection. Jaegle et al. (2021) uses these latent arrays to compress high-dimensional multimodal inputs

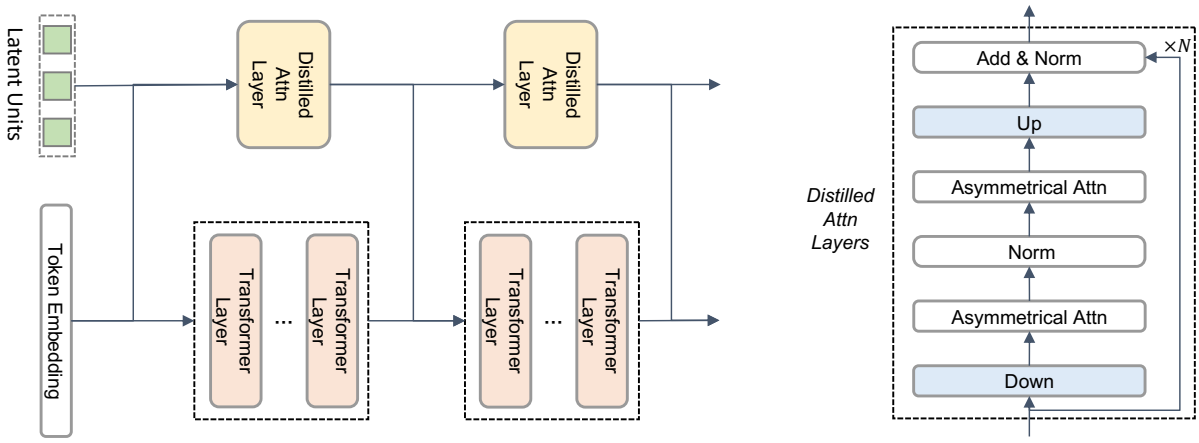


Figure 3: The overall architecture of the proposed Global Controller. On the left, we use a set of latent units as the global controller that runs through all layers of the backbone model to steer the capabilities of LLMs from a global perspective. These latent units iteratively distill information features from the LLM and update the hidden state of the LLM using the distilled attention mechanism. On the right, we show the various modules of the distilled attention mechanism. We first map the hidden state dimensions to lower dimensions, after which we execute the asymmetric attention mechanism on the lower dimensions to facilitate bidirectional interactions between latent units and freezed representations, reducing the training memory while mitigating the problem of non-full-rank training.

and refers to them as latent units. Recently, Wang et al. (2023b) uses these latent variables in context learning, which is utilized to model latent topics or concepts as:

$$P\left(\boldsymbol{x}_{1:T}\right)=\int_{\Theta} P\left(\boldsymbol{x}_{1:T} \mid \boldsymbol{\theta}\right) P(\boldsymbol{\theta}) d \boldsymbol{\theta}, \quad (3)$$

Where $\boldsymbol{\theta} \in \Theta$ represents a latent high dimensional topic or concept variable, Θ is the space of the topic or concept variable, and $\boldsymbol{x}_{1:T}$ refers to the input token embedding.

3.2 Overview

As shown in Figure 3, we treats LLM as a feature extractor and uses a set of latent units $U = \mathbb{R}^{M\times d}$ to perform the exchange of information with LLM. The number of the units M is pre-specified. By the nature of the small and independent nature of the latent units relative to the size of the input, this greatly reduces the requirement for training memory. Formally, for L transformer layers, we use a control factor s for chunking, where s can be divisible by L. After that, for each transformer chunk, we will use a distilled attention mechanism layer to complete the information interaction between latent units and hidden states, where the number of transformer chunks, i.e., the number of distilled attention mechanism layers, is $N = L/s$. When $N = L$, it means that we exchange information at each transformer layer, but this leads to an increase in computation and training memory. We denote the latent units U transformed by the i-th distilled attention layer as $u_i \in \mathbb{R}^{M\times d}$, and similarly, the input embedding X transformed by the i-th transformer block as $h_i \in \mathbb{R}^{n\times d}$. Afterwards, we employ asymmetric attentional mechanisms to facilitate bidirectional interactions between latent units u_i and freezed representations of the backbone model h_i. Since we replace simple summation with bidirectional attentional interactions, we alleviate the problem of training with non-full-rank training. Further, we apply a distillation technique to the hidden states during the interaction to compress the hidden state size to a very small scale, which ensures fewer trainable parameters.

It is worth mentioning that, our approach can be framed as prefix fine-tuning with low-rank matrices constrained by masking. Ding et al. (2023a) mentioned the instability issue in prefix training, which we attribute to the strong coupling between the prefix units and backbone models without additional mapping matrices and masking, resulting in a large difference in the initial variable space. Compared to prefix-tuning, our method has an additional low-rank mapping space (the former two use the QKV mapping of the backbone model itself), which leads to better training convergence while mitigating expressive bottlenecks.

3.3 Distilled Attention

Based on Jaegle et al. (2021), we design distilled attention mechanisms for parameter-efficient training.

Algorithm 1: Distilled Attention

Input: Latent units u_i, token hidden states h_i, down projection weights D_u, D_h, up projection weights P_u, P_h

begin

$u'_i, h'_i \leftarrow u_i D_u, h_i D_h$; // down proj

$u'_i \leftarrow \mathrm{MHA}(u'_i, h'_i, h'_i)$; // attention

$u'_i \leftarrow \mathrm{LN}(u'_i)$; // layer norm

$h'_i \leftarrow \mathrm{MHA}(h'_i, u'_i, u'_i)$; // attention

$u''_i, h''_i \leftarrow u'_i P_u, h'_i P_h$; // up proj

$h''_i \leftarrow \mathrm{LN}(h''_i)$; // layer norm

$u_i, h_i \leftarrow u_i + u''_i, h_i + h''_i$; // residual

Output: u_i, h_i

Different from Jaegle et al. (2021) which predicts the labeling probability of downstream tasks based on latent arrays, we use the asymmetric attention mechanism to perform the information exchange between the latent units and the backbone model, and ultimately generate the probability distribution based on the hidden state of the backbone model. As shown in Algorithm 1, we design the distillation module to project the latent units u_i and hidden state dimensions h_i to lower dimensions as u'_i, h'_i, thus satisfying the need for parameter-efficient training.

Asymmetrical attention We build our information exchange architecture around the attention mechanism because it is both universally applicable and powerful in practice. The main challenge facing traditional attention is that the complexity of Q-K-V self-attention is quadratic in the number of input dimensions, while the length n of the input sequence is usually very large. Here, we apply attention directly to M latent units by introducing asymmetry in the attention operation. The resulting attention operation has complexity $\mathcal{O}(Mn)$. Since the number of latent cells is much smaller than the length of the input sequence ($M \ll n$, e.g., $M = 8$ when $n = 512$), this greatly reduces the computational complexity. In addition, since we use the bidirectional attentional interaction mechanism instead of simple summation in the traditional approach, this helps alleviate the problem of non-full-rank training (He et al., 2022a).

Projection It is worth noting that the inclusion of the Q-K-A mapping matrices in the traditional MHA module introduces a large number of parametric quantities, which is not desired in parameter-efficient fine-tuning methods. In order to reduce the introduction of such parameters, we draw on the technique of distillation (Zhao et al., 2019) that projects the latent units u_i and hidden state dimensions h_i to lower dimensions as u'_i, h'_i. Formally, we denote the down and up projection matrices as $D_u, D_h \in \mathbb{R}^{d \times d'}$ and $P_u, P_h \in \mathbb{R}^{d' \times d}$. As shown in Algorithm 1, we execute the asymmetric attention mechanism in lower dimensions ($d' \ll d$, e.g., $d' = 8$ when $d = 768$), which further reduces the number of computational parameters.

Dataset	#Train	#Valid	#Test	Metric
CoLA	8.5k	1,043	1,063	Mcc
SST-2	67k	872	1.8k	Acc
MRPC	3.7k	408	1.7k	Acc
QQP	364k	40.4k	391k	Acc/F1
STS-B	5.7k	1.5k	1.4k	Corr
MNLI	393k	9.8k/9.8k	9.8k/9.8k	Acc(m/mm)
QNLI	105k	5.5k	5.5k	Acc
RTE	2.5k	277	3k	Acc

Table 1: Dataset statistics and metric in GLUE benchmark. "Mcc", "Acc", "F1" and "Corr" represent matthews correlation coefficient, accuracy, the F1 score and pearson correlation coefficient respectively.

4 Experiments

To demonstrate the effectiveness of the proposed global controller method (GloC), we conduct extensive experiments on a range of natural language understanding, generation, and reasoning tasks.

4.1 Datasets

For evaluation on natural language understanding and generation tasks, we adopt the GLUE benchmark (Wang et al.), including CoLA (Warstadt et al., 2019), SST-2 (Socher et al., 2013), MRPC (Dolan and Brockett, 2005), QQP (Wang et al.), STS-B (Wang et al.), MNLI (Williams et al., 2018), QNLI (Rajpurkar et al., 2016) and RTE (Dagan et al., 2005; Haim et al., 2006; Giampiccolo et al., 2007; Bentivogli et al., 2009) Table 1 shows the detailed dataset statistics and the evaluation metric. Additionally, for the reasoning task, we take six common-sense reasoning datasets, including BoolQ (Clark et al., 2019), PIQA (Bisk et al., 2020), SIQA (Sap et al., 2019), etc. BoolQ is a question-answering dataset for yes/no questions containing 15942 examples. These problems occur and arise in unprompted and unconstrained environments. PIQA consists of questions with two solutions requiring physical commonsense to an-

Method	#Params	CoLA	SST-2	MRPC	QQP	STS-B	MNLI	QNLI	RTE
Fine-Tune	184M	69.21	95.64	89.22	92.05/89.31	91.59	89.98/89.95	93.78	82.49
Adapter	1.41M	69.00	95.16	89.90	91.45/88.88	92.21	90.11/90.11	93.79	82.44
Bitfit	0.1M	68.70	94.38	87.16	87.86/84.20	89.71	87.45/87.45	91.90	76.12
LoRA (r=8)	1.33M	69.73	95.57	89.71	91.95/89.26	91.86	90.47/90.46	93.76	85.32
AdaLoRA	1.27M	70.86	95.95	90.22	92.13/88.41	91.39	90.27/90.30	94.28	87.36
SoRA	0.91M	71.48	95.64	91.98	**92.39/89.87**	92.22	90.35/90.38	**94.28**	87.77
GloC	1.33M	**72.35**	**96.16**	**92.31**	92.36/89.54	**92.45**	**90.62/90.45**	94.15	**88.36**

Table 2: Test results of the proposed method and other baselines on the GLUE benchmark. We denote the best result in **bold**. We report mean of 5 runs using different random seeds.

swer. SIQA focuses on reasoning about people's actions and their social implications. The detailed dataset descriptions and data statistics are provided in the Appendix A.1.

4.2 Implementation Details

For fair comparison with prior work, we use DeBERTaV3-base (He et al., 2021) as the backbone model for the GLUE benchmark, Llama (Touvron et al., 2023b) and Llama2 (Touvron and et al., 2023) for the common-sense reasoning datasets. For both versions of a range of sizes of Llama, we adopt 7B size. Commonly, we use the AdamW (Loshchilov and Hutter, 2017) optimizer with a linear warmup-decay learning schedule and a dropout (Srivastava et al., 2014) of 0.1. The latent units are randomly initialized with the normal distribution $\mathcal{N}(0.0, 0.02)$. For hyper-parameters, we set the learning rate to 1e-4 with a batch size of 32. In the main experiments, if not additionally mentioned, we set $M = 8$, $s = 2$ and $d' = 16$. For almost all experiments, we run 5 times using different random seeds and report the average results in order to ensure statistical significance.

4.3 Baselines

In this paper, we compare the proposed GloC with full-parameter fine-tuning and the following robust baseline models: Adapter (Houlsby et al., 2019), BitFit (Zaken et al., 2021), LoRA (Hu et al., 2021), AdaLoRA (Zhang et al., 2023) and SoRA (Ding et al., 2023a). Notably, on the common-sense reasoning task, since full parameter fine-tuning of a large model is unaffordable for a small workshop, we use the results of ChatGPT[1] as an alternative to full fine-tuning. We use GPT-3.5 `text-Davinci-003` for Zero-shot CoT (Kojima et al., 2022) as the baseline.

4.4 Overall Performance

GLUE performance As shown in Table 2, SoRA and AdaLoRA, the state-of-the-art methods on the current ranking, outperform previous methods, including LoRA, on a range of understanding and generation tasks, demonstrating the effectiveness of dynamically adjusting the rank. More evidently, our proposed method achieves state-of-the-art results on six subtasks and comparable results on two others. As an example, GloC outperforms AdaLoRA by 1.49% and 2.09% on CoLA and MRPC, respectively, which demonstrates the effectiveness of our global perspective. We consider the set of latent units as a global controller that runs through all layers of the large language model, fully exploiting the capabilities of LLM.

Reasoning performance On the average F1-measure of the 6 common-sense datasets in Table 7, the result of AdaLoRA (Zhang et al., 2023) improves over the LoRA baseline by +0.4% and +0.7% on different Llama version, while our method further improves by +2.2% and +2.0% compared to AdaLoRA, which speaks volumes about the effectiveness of our approach. In addition, our method outperforms ChatGPT by 3.3% and 3.7% on HellaS and OBQA, respectively, which suggests that parameter-efficient fine-tuning methods still have a lot of potential and room for development when compared to state-of-the-art LLMs. We attribute these performance improvements to that we employ asymmetric attentional mechanisms to facilitate bidirectional interactions between latent units and freezed representations, hence mitigating the problems associated with non-full-rank training. In addition, we believe that the latent units

[1] `https://openai.com/blog/chatgpt/`

Method	# Params	BoolQ	PIQA	SIQA	HellaS	WinoG	OBQA	AVE.
ChatGPT	-	73.1	85.4	68.5	78.5	66.1	74.8	74.4
LLaMA-Adapter	0.77%	67.9	76.4	78.8	69.8	78.9	75.2	74.5
LLaMA-LoRA	0.72%	68.9	80.7	77.4	78.1	76.8	74.8	76.1
LLaMA-AdaLora	0.69%	69.4	80.8	77.8	78.6	77.1	75.4	76.5
LLaMA-GloC	0.72%	72.1	83.6	78.7	81.4	79.2	77.1	78.7
LLaMA2-Adapter	0.77%	68.2	78.1	78.4	71.2	78.1	75.6	74.9
LLaMA2-LoRA	0.72%	70.8	82.4	78.8	78.5	77.4	74.8	77.1
LLaMA2-AdaLora	0.69%	71.2	82.9	78.8	79.6	77.8	76.6	77.8
LLaMA2-GloC	0.72%	**73.8**	**85.3**	**79.1**	**81.8**	**80.4**	**78.5**	**79.8**

Table 3: Results with LLaMA & LLaMA2 on six common-sense reasoning datasets. The best results on each dataset are shown in **bold**. We report mean of 5 runs using different random seeds.

Method	CoLA	STS-B	QNLI
Default	**72.35**	92.45	**94.15**
freezed units	72.24	92.28	94.06
one-way interaction	70.96	90.32	93.57
w/o projection	72.35	**92.62**	94.09
w/ FFN	71.86	90.85	93.64

Table 4: Ablation studies with four different settings of our method on three GLUE datasets.

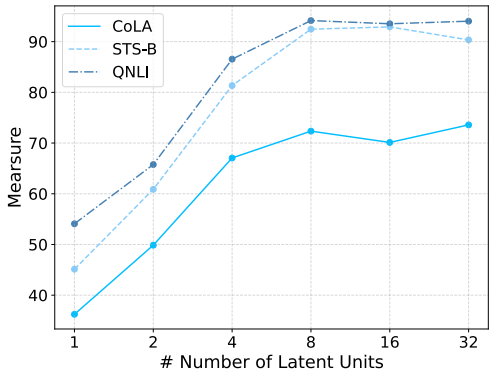


Figure 4: The performance of the model under different number of latent units on the three GLUE datasets.

learn task-specific linguistic information that further improves the performance of the model. We will conduct extensive ablation experiments in subsequent sections to support our points. We also supplement the results of prefix and prompt tuning in the Appendix A.2. More analysis for non-full-rank training can be found in the Appendix A.3.

Another thing to keep in mind is that, we find in practice that GloC has slightly more parametric quantities than LoRA, due to the fact that in addition to the parameters of the projection, the latent cells also take up part of the parametric quantities. We ensure that the number of parameters is basically the same as LoRA by controlling the chunking factor s and the low-dimensional size d' of the projection in our experiments, e.g., when r is 8 in LoRA, we set $s = 2$ and $d' = 16$. We will carefully analyze the impact of these hyperparameters on the model performance.

5 Analysis

5.1 Ablation Study

In this section, we perform extensive ablation experiments to analyze the necessity of each sub-module of the proposed method and the extent of its impact on the performance. We mainly design the following four setup: (1) **freezed units**: we freeze the feature space of latent cells and learn only the projection matrices in the experiments, (2) **one-way interaction**: we set up so that latent units no longer draw information from the backbone model, turning what was originally a bidirectional interaction into a unidirectional one, (3) **w/o projection**: we no longer map the features in low dimensions, keeping the mapping matrices as in the original MHA, which would add much training time, (4) **w/ FFN**: based on (3), we add two additional layers of FFNs after MHA, which leads to a further expansion of the number of parameters.

From Table 4 we observe that **one-way interaction** leads to a great significant performance decrease in absolute acc-mearsure (-1.39% and -2.13% on CoLA and STS-B), which demonstrates the effectiveness of the bidirectional interaction we set. We argue that the asymmetric attentional mechanisms between latent units and freezed representations mitigates the problems associated with

Factor s	Dim d'	CoLA	STS-B	QNLI	# Params
$s=1$	$d'=8$	72.26	91.97	94.02	1×
	$d'=16$	72.29	92.24	94.08	2×
$s=2$	$d'=16$	72.35	**92.45**	94.15	1×
	$d'=32$	**72.44**	92.36	**94.15**	2×
$s=4$	$d'=16$	72.16	91.89	93.74	0.5×
	$d'=32$	72.25	92.30	94.12	1×
$s=6$	$d'=24$	71.84	91.62	93.47	0.5×
	$d'=48$	72.17	92.18	93.85	1×

Table 5: The effect of different control chunking factors s as well as low dimensional mapping sizes d' on model performance. $1\times$ denotes the default model, which is essentially comparable to the parameters of the LoRA model with rank 8.

non-full-rank training, leading to promising performance. In addition, as show in Table 4, freezing the learning space of latent units can also be slightly detrimental to model performance (-0.11% and -0.17% in absolute acc-mearsure). This suggests that the units have also learned some information that is helpful for the model. We will analyze later if this information is relevant to the specific task.

However, **w/o projection** that not doing low-dimensional mapping does not significantly increase the performance of the model, and the results on CoLA do not change. This suggests that our mapping approach does not harm the model performance with reduced model parameters. We tried adding additional FFN modules and instead observed a decrease in model performance (-1.6% on STS-B and -0.51% on QNLI). This suggests that not more parameters lead to better performance. This is in line with the original intent of a series of parameter-efficient fine-tuning methods (He et al., 2022a; Zhang et al., 2023).

5.2 Analysis of Hyper-parameters

In this section, we focus on analyzing the impact of different hyperparameters on the model performance, including the number of latent units M, the factor controlling the chunking s, and the low-dimensional size of the projection d'.

Number of latent units Since the number of latent units is pre-fixed, we conduct a comparison experiment to find the suitable setting. We employ the experiment on three GLUE datasets with a range from 1 to 32. We can observe from Figure 4 that the model performance increases as the latent units grows. However, when the number of the latent units 8, the trend of performance improvement on all three datasets slows down, or even decreases slightly, suggesting that for these current sub-tasks, the representational power of 8 latent units is sufficient for modeling the relevant information needed for the task. Eventually, the latent units number M in the experiments is set to 8, which is deployed on all the other datasets.

Datasets	**AdaLoRA (s)**	**SoRA (s)**	**GloC (s)**
CoLA	160.2	57.2	110.4
SST-2	491.0	433.0	453.6
MRPC	27.3	24.8	26.2
STS-B	48.2	38.4	40.3
QNLI	1001.0	676.3	738.0
RTE	79.8	45.1	64.4
Avg.	301.3	212.5	238.8

Table 6: The average training time per epoch on six datasets. For each task, all experiments have the same batch size 32.

Chunking and hidden size To be explicit, excluding the number of parameters occupied by latent units, the number of model parameters stays the same when the rank of the LoRA model is $r = d'/s$. We observe in Table 5 that the results in the second row are generally higher than in the first row, suggesting that a larger d' leads to better model performance. The factor s controlling the chunking achieves the best results for a value of 2, suggesting that sparser information exchange leads to a decrease in model performance. For example, when $s = 6$, it brings a performance degradation of -0.27% on CoLA and -0.3% on QNLI, respectively.

5.3 Analysis of Training Cost

In addition to the additional introduction of the number of parameters, the state-of-art method (Zhang et al., 2023; Ding et al., 2023a) also takes into account the cost of training time. The results in Table 2 show that the proposed method achieves superior performance against LoRA and other methods which basically have the same number of parameters. From the results in the Table 6, the training time of the proposed method is comparable to other advanced methods. AdaLoRA Zhang et al. (2023) computing SVD introduces additional computational overhead, and SoRA's well-designed training schedule Ding et al. (2023a) leads to shorter time.

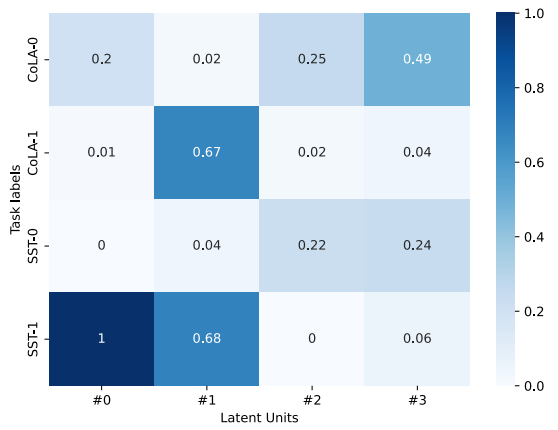


Figure 5: Co-occurrence statistics between the latent units and different task labels.

5.4 Analysis of Latent Units

Previous ablation experiments verified that the latent units learned features that contribute to the model's performance, and in this section we wish to verify that the latent units learned task-specific features, such as task topic content or task labeling information.

We count the co-occurrence of different latent units and different task labels on CoLA and SST-2 datasets. To eliminate the imbalance, we normalize the co-occurrence matrix first. As shown in Figure 5, different latent units have preferences for different task labels. For example, on CoLA task, the latent units #1 prefer to predict `CoLA-1` labels, while on SST-2 task, the latent units #2 and # 3 prefer to predict `SST-0` labels. These findings indicate that the latent units show strong statistical correlations with task labels, suggesting that the latent units learn task-specific relevant features.

6 Conclusion

In this paper, we present a novel global controller (GloC) approach for parameter-efficient fine-tuning. Based on a small set of latent units, we harness the large language model from a global perspective to seek optimal performance. We design asymmetric attention mechanisms and distillation compression modules during interaction to reduce training memory while mitigating the problem of non-full-rank training. Extensive experiments on a range of natural language understanding, generation, and reasnoning tasks show that our model reaches the state-of-the-art and significantly outperforms a range of robust baselines.

Limitations

We discuss here the limitations of the method in this paper. First, the proposed method has some hyperparameters, such as the number of latent units and the projection size. When the method needs to be migrated to a new task, parameter search is inevitably required. The second point is that the proposed methods have not been validated on ultra-large scale macromodels, such as Llama-70B, which we will validate in the subsequent work. The third point is that all PEFT methods inevitably have training memory bottlenecks due to the need for forward passes in the backbone model, and exploring new migration methods is a valuable direction.

Acknowledgments

This work is supported by the National Natural Science Foundation of China (No. 62376245), the Key Research and Development Program of Zhejiang Province, China (No. 2024C03255), the Fundamental Research Funds for the Central Universities, and National Key Research and Development Project of China (No. 2018AAA0101900).

References

Luisa Bentivogli, Peter Clark, Ido Dagan, and Danilo Giampiccolo. 2009. The fifth pascal recognizing textual entailment challenge. In *Proceedings of Text Analysis Conference*.

Srinadh Bhojanapalli, Chulhee Yun, Ankit Singh Rawat, Sashank Reddi, and Sanjiv Kumar. 2020. Low-rank bottleneck in multi-head attention models. In *International conference on machine learning*, pages 864–873. PMLR.

Yonatan Bisk, Rowan Zellers, Ronan Le Bras, Jianfeng Gao, and Yejin Choi. 2020. Piqa: Reasoning about physical commonsense in natural language. In *Thirty-Fourth AAAI Conference on Artificial Intelligence*.

Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*.

Nicolas Carion, Francisco Massa, Gabriel Synnaeve, Nicolas Usunier, Alexander Kirillov, and Sergey Zagoruyko. 2020. End-to-end object detection with transformers.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, and others. 2022. Palm:

P-Tuning v2: Prompt Tuning Can Be Comparable to Fine-tuning Universally Across Scales and Tasks

Xiao Liu[1,2*], **Kaixuan Ji**[1*], **Yicheng Fu**[1*], **Weng Lam Tam**[1], **Zhengxiao Du**[1,2], **Zhilin Yang**[1,3†], **Jie Tang**[1,2†]
[1]Tsinghua University, KEG [2]Beijing Academy of Artificial Intelligence (BAAI)
[3]Shanghai Qi Zhi Institute
{liuxiao21,jkx19,fyc19}@mails.tsinghua.edu.cn

Abstract

Prompt tuning, which only tunes continuous prompts with a frozen language model, substantially reduces per-task storage and memory usage at training. However, in the context of NLU, prior work reveals that prompt tuning does not perform well for normal-sized pretrained models. We also find that existing methods of prompt tuning cannot handle hard sequence labeling tasks, indicating a lack of universality. We present a novel empirical finding that properly optimized prompt tuning can be universally effective across a wide range of model scales and NLU tasks. It matches the performance of finetuning while having only 0.1%-3% tuned parameters. Our method P-Tuning v2 is an implementation of Deep Prompt Tuning (Li and Liang, 2021; Qin and Eisner, 2021) optimized and adapted for NLU. Given the universality and simplicity of P-Tuning v2, we believe it can serve as an alternative to finetuning and a strong baseline for future research.[1]

1 Introduction

Pretrained language models (Radford et al., 2019; Devlin et al., 2018; Yang et al., 2019; Raffel et al., 2019) improve performance on a wide range of natural language understanding (NLU) tasks. A widely-used method, **fine-tuning**, updates the entire set of model parameters for a target task. While fine-tuning obtains good performance, it is memory-consuming during training because gradients and optimizer states for all parameters must be stored. Moreover, keeping a copy of model parameters for each task during inference is inconvenient since pre-trained models are usually large.

Prompting, on the other hand, freezes all parameters of a pre-trained model and uses a natural language prompt to query a language model (Brown et al., 2020). For example, for sentiment analysis, we can concatenate a sample (e.g., "Amazing movie!") with a prompt "This movie is [MASK]" and ask the pre-trained language model to predict the probabilities of masked token being "good" and "bad" to decide the sample's label. Prompting requires no training at all and stores one single copy of model parameters. However, discrete prompting (Shin et al., 2020; Gao et al., 2020) can lead to suboptimal performance in many cases compared to fine-tuning.

Figure 1: Average scores on RTE, BoolQ and CB of SuperGLUE dev. With 0.1% task-specific parameters, P-tuning v2 can match fine-tuning across wide scales of pre-trained models, while Lester et al. (2021) & P-tuning can make it conditionally at 10B scale.

Prompt tuning[2] is an idea of tuning only the continuous prompts. Specifically, Liu et al. (2021); Lester et al. (2021) proposed to add trainable continuous embeddings (also called continuous prompts) to the original sequence of input word embeddings. Only the continuous prompts are updated during training. While prompt tuning improves over prompting on many tasks (Liu et al., 2021; Lester et al., 2021; Zhong et al., 2021), it still underperforms fine-tuning when the model size is not large, specifically less than 10 billion parameters (Lester et al., 2021). Moreover, as shown in our experiments, prompt tuning performs poorly compared to fine-tuning on several hard sequence labeling tasks such as extractive question answer-

† corresponding to: Zhilin Yang (zhiliny@tsinghua.edu.cn) and Jie Tang (jietang@tsinghua.edu.cn)
* indicates equal contribution.
[1]Our code and data are released at https://github.com/THUDM/P-tuning-v2.
[2]We use "prompt tuning" to refer to a class of methods rather than a particular method.

ing (Cf. Section 4.2).

Our main contribution in this paper is a novel empirical finding that properly optimized prompt tuning can be comparable to fine-tuning universally across various model scales and NLU tasks. In contrast to observations in prior work, our discovery reveals the universality and potential of prompt tuning for NLU.

Technically, our approach P-tuning v2 is not conceptually novel. It can be viewed as an optimized and adapted implementation of **Deep Prompt Tuning** (Li and Liang, 2021; Qin and Eisner, 2021) designed for generation and knowledge probing. The most significant improvement originates from appling continuous prompts for every layer of the pretrained model, instead of the mere input layer. Deep prompt tuning increases the capacity of continuous prompts and closes the gap to fine-tuning across various settings, especially for small models and hard tasks. Moreover, we present a series of critical details of optimization and implementation to ensure finetuning-comparable performance.

Experimental results show that P-tuning v2 matches the performance of fine-tuning at different model scales ranging from 300M to 10B parameters and on various hard sequence tagging tasks such as extractive question answering and named entity recognition. P-tuning v2 has 0.1% to 3% trainable parameters per task compared to fine-tuning, which substantially reduces training time memory cost and per-task storage cost.

2 Preliminaries

NLU Tasks. In this work, we categorize NLU challenges into two families: *simple classification tasks* and *hard sequence labeling tasks*.[3] Simple classification tasks involve classification over a label space. Most datasets from GLUE (Wang et al., 2018) and SuperGLUE (Wang et al., 2019) are in this category. Hard sequence labeling tasks involve classification over a sequence of tokens, such as named entity recognition and extractive question answering.

Prompt Tuning. Let $\mathcal{V}$ be the vocabulary of a language model $\mathcal{M}$ and let $\mathbf{e}$ be the embedding layer of $\mathcal{M}$. In the case of discrete prompting (Schick and Schütze, 2020), prompt tokens {"It", "is", "[MASK]"} $\subset$ $\mathcal{V}$ can be used to classify a movie review. For example, given the input text $\mathbf{x}$ ="Amazing movie!", the input embedding sequence is formulated as $[\mathbf{e}(\mathbf{x}), \mathbf{e}(\text{"It"}), \mathbf{e}(\text{"is"}), \mathbf{e}(\text{"[MASK]"})]$.

Lester et al. (2021) and Liu et al. (2021) introduce trainable continuous prompts as a substitution to natural language prompts for NLU with the parameters of pretrained language models frozen. Given the trainable continuous embeddings $[h_0, ..., h_i]$, the input embedding sequence is written as $[\mathbf{e}(\mathbf{x}), h_0, ..., h_i, \mathbf{e}(\text{"[MASK]"})]$, as illustrated in Figure 2. Prompt tuning has been proved to be comparable to fine-tuning on 10-billion-parameter models on simple classification tasks (Lester et al., 2021; Kim et al., 2021; Liu et al., 2021).

3 P-Tuning v2

3.1 Lack of Universality

Lester et al. (2021); Liu et al. (2021) have been proved quite effective in many NLP applications (Wang et al., 2021a,b; Chen et al., 2021; Zheng et al., 2021; Min et al., 2021), but still fall short at replacing fine-tuning due to lack of universality, as discussed below.

Lack of universality across scales. Lester et al. (2021) shows that prompt tuning can be comparable to fine-tuning when the model scales to over 10 billion parameters. However, for medium-sized models (from 100M to 1B) that are widely used, prompt tuning performs much worse than fine-tuning.

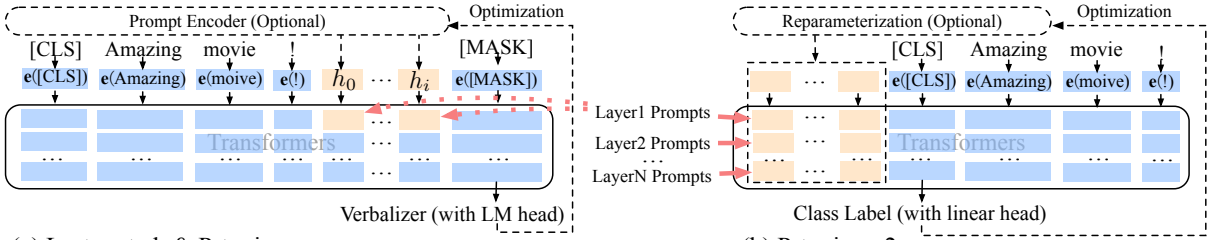
Lack of universality across tasks. Though Lester et al. (2021); Liu et al. (2021) have shown superiority on some of the NLU benchmarks, the effectiveness of prompt tuning on hard sequence tagging tasks is not verified. Sequence tagging predicts a sequence of labels for each input token, which can be harder and incompatible with verbalizers (Schick and Schütze, 2020). In our experiments (Cf. Section 4.2 and Table 3), we show that Lester et al. (2021); Liu et al. (2021) perform poorly on typical sequence tagging tasks compared to fine-tuning.

Considering these challenges, we propose P-tuning v2, which adapts deep prompt tuning (Li and Liang, 2021; Qin and Eisner, 2021) as a universal solution across scales and NLU tasks.

3.2 Deep Prompt Tuning

In (Lester et al., 2021) and (Liu et al., 2021), continuous prompts are only inserted into the input

[3] Note that the notions of "simple" and "hard" are specific to prompt tuning, because we find sequence labeling tasks are more challenging for prompt tuning.



(a) Lester et al. & P-tuning (Frozen, 10-billion-scale, simple tasks) (b) P-tuning v2 (Frozen, most scales, most tasks)

Figure 2: From Lester et al. (2021) & P-tuning to P-tuning v2. Orange blocks (i.e., $h_0, ..., h_i$) refer to trainable prompt embeddings; blue blocks are embeddings stored or computed by frozen pre-trained language models.

embedding sequence (Cf. Figure 2 (a)). This leads to two challenges. First, the number of tunable parameters is limited due to the constraints of sequence length. Second, the input embeddings have relatively indirect impact on model predictions.

To address these challenges, P-tuning v2 employs the idea of deep prompt tuning (Li and Liang, 2021; Qin and Eisner, 2021). As illustrated in Figure 2, prompts in different layers are added as prefix tokens. On one hand, P-tuning v2 have more tunable task-specific parameters (from 0.01% to 0.1%-3%) to allow more per-task capacity while being parameter-efficient; on the other hand, prompts added to deeper layers have more direct impact on model predictions (see analysis in Appendix B).

3.3 Optimization and Implementation

There are a few useful details of optimization and implementation for achieving the best performance.

Reparameterization. Prior works usually leverage a reparameterization encoder such as an MLP (Li and Liang, 2021; Liu et al., 2021) to transform trainable embeddings. However, for NLU, we discover that its usefulness depends on tasks and datasets. For some datasets (e.g., RTE and CoNLL04), MLP brings a consistent improvement; for the others, MLP leads to minimal or even negative effects on the results (e.g., BoolQ and CoNLL12). See Appendix B for more analysis.

Prompt Length. The prompt length plays a critical role in P-Tuning v2. We find that different NLU tasks usually achieve their best performance with different prompt lengths (Cf. Appendix B). Generally, simple classification tasks prefer shorter prompts (less than 20); hard sequence labeling tasks prefer longer ones (around 100).

Multi-task Learning. Multi-task learning jointly optimizes multiple tasks with shared continuous prompts before fine-tuning for individual tasks.

Method	Task	Re-param.	Deep PT	Multi-task	No verb.
P-tuning (Liu et al., 2021)	KP NLU	LSTM	-	-	-
PROMPTTUNING (Lester et al., 2021)	NLU	-	-	✓	-
Prefix Tuning (Li and Liang, 2021)	NLG	MLP	✓	-	-
SOFT PROMPTS (Qin and Eisner, 2021)	KP	-	✓	-	-
P-tuning v2 (Ours)	NLU SeqTag	*(depends)*	✓	✓	✓

Table 1: Conceptual comparison between P-tuning v2 and existing Prompt Tuning approaches (KP: Knowledge Probe; SeqTag: Sequence Tagging; Re-param.: Reparameterization; No verb.: No verbalizer).

Multi-task is optional for P-Tuning v2 but can be used for further boost performance by providing a better initialization (Gu et al., 2021).

Classification Head. Using a language modeling head to predict verbalizers (Schick and Schütze, 2020) has been central for prompt tuning (Liu et al., 2021), but we find it unnecessary in a full-data setting and incompatible with sequence labeling. P-tuning v2 instead applies a randomly-initialized classification head on top of the tokens as in BERT (Devlin et al., 2018) (Cf. Figure 2).

To clarify P-tuning v2's major contribution, we present a conceptual comparison to existing prompt tuning approaches in Table 1.

4 Experiments

We conduct extensive experiments over different commonly-used pre-trained models and NLU tasks to verify the effectiveness of P-tuning v2. In this work, all methods except for fine-tuning are conducted with **frozen language model backbones**, which accords with (Lester et al., 2021)'s setting but differs from (Liu et al., 2021)'s tuned setting.

	#Size	BoolQ			CB			COPA			MultiRC (F1a)		
		FT	PT	PT-2	FT	PT	PT-2	FT	PT	PT-2	FT	PT	PT-2
$\text{BERT}_{\text{large}}$	335M	**77.7**	67.2	75.8	**94.6**	80.4	**94.6**	69.0	55.0	**73.0**	70.5	59.6	**70.6**
$\text{RoBERTa}_{\text{large}}$	355M	**86.9**	62.3	84.8	98.2	71.4	**100**	**94.0**	63.0	93.0	**85.7**	59.9	82.5
$\text{GLM}_{\text{xlarge}}$	2B	**88.3**	79.7	87.0	**96.4**	76.4	**96.4**	**93.0**	92.0	91.0	84.1	77.5	**84.4**
$\text{GLM}_{\text{xxlarge}}$	10B	88.7	**88.8**	**88.8**	**98.7**	98.2	96.4	**98.0**	**98.0**	**98.0**	**88.1**	86.1	**88.1**

	#Size	ReCoRD (F1)			RTE			WiC			WSC		
		FT	PT	PT-2	FT	PT	PT-2	FT	PT	PT-2	FT	PT	PT-2
$\text{BERT}_{\text{large}}$	335M	70.6	44.2	**72.8**	70.4	53.5	**78.3**	74.9	63.0	**75.1**	**68.3**	64.4	**68.3**
$\text{RoBERTa}_{\text{large}}$	355M	89.0	46.3	**89.3**	86.6	58.8	**89.5**	**75.6**	56.9	73.4	63.5	**64.4**	63.5
$\text{GLM}_{\text{xlarge}}$	2B	91.8	82.7	**91.9**	**90.3**	85.6	**90.3**	**74.1**	71.0	72.0	**95.2**	87.5	92.3
$\text{GLM}_{\text{xxlarge}}$	10B	**94.4**	87.8	92.5	**93.1**	89.9	**93.1**	**75.7**	71.8	74.0	**95.2**	94.2	93.3

Table 2: Results on SuperGLUE development set. P-tuning v2 surpasses P-tuning & Lester et al. (2021) on models smaller than 10B, matching the performance of fine-tuning across different model scales. (FT: fine-tuning; PT: Lester et al. (2021) & P-tuning; PT-2: P-tuning v2; **bold**: the best; underline: the second best).

	#Size	CoNLL03				OntoNotes 5.0				CoNLL04			
		FT	PT	PT-2	MPT-2	FT	PT	PT-2	MPT-2	FT	PT	PT-2	MPT-2
$\text{BERT}_{\text{large}}$	335M	**92.8**	81.9	90.2	91.0	**89.2**	74.6	86.4	86.3	85.6	73.6	84.5	**86.6**
$\text{RoBERTa}_{\text{large}}$	355M	92.6	86.1	**92.8**	**92.8**	**89.8**	80.8	**89.8**	**89.8**	88.8	76.2	88.4	**90.6**
$\text{DeBERTa}_{\text{xlarge}}$	750M	**93.1**	90.2	**93.1**	**93.1**	90.4	85.1	90.4	**90.5**	89.1	82.4	86.5	**90.1**

	#Size	SQuAD 1.1 dev (EM / F1)								SQuAD 2.0 dev (EM / F1)							
		FT		PT		PT-2		MPT-2		FT		PT		PT-2		MPT-2	
$\text{BERT}_{\text{large}}$	335M	**84.2**	**91.1**	1.0	8.5	77.8	86.0	82.3	89.6	**78.7**	**81.9**	50.2	50.2	69.7	73.5	72.7	75.9
$\text{RoBERTa}_{\text{large}}$	355M	**88.9**	**94.6**	1.2	12.0	88.5	94.4	88.0	94.1	**86.5**	**89.4**	50.2	50.2	82.1	85.5	83.4	86.7
$\text{DeBERTa}_{\text{xlarge}}$	750M	90.1	95.5	2.4	19.0	**90.4**	**95.7**	89.6	95.4	88.3	91.1	50.2	50.2	**88.4**	**91.1**	88.1	90.8

	#Size	CoNLL12				CoNLL05 WSJ				CoNLL05 Brown			
		FT	PT	PT-2	MPT-2	FT	PT	PT-2	MPT-2	FT	PT	PT-2	MPT-2
$\text{BERT}_{\text{large}}$	335M	84.9	64.5.	83.2	**85.1**	**88.5**	76.0	86.3	**88.5**	82.7	70.0	80.7	**83.1**
$\text{RoBERTa}_{\text{large}}$	355M	**86.5**	67.2	84.6	86.2	**90.2**	76.8	89.2	90.0	85.6	70.7	84.3	**85.7**
$\text{DeBERTa}_{\text{xlarge}}$	750M	86.5	74.1	85.7	**87.1**	**91.2**	82.3	90.6	**91.2**	86.9	77.7	86.3	**87.0**

Table 3: Results on Named Entity Recognition (NER), Question Answering (Extractive QA), and Semantic Role Labeling (SRL). All metrics in NER and SRL are micro-f1 score. (FT: fine-tuning; PT: P-tuning & Lester et al. (2021); PT-2: P-tuning v2; MPT-2: Multi-task P-tuning v2; **bold**: the best; underline: the second best).

Ratios of task-specific parameters (e.g., 0.1%) are derived from comparing continuous prompts' parameters with transformers' parameters. Another thing to notice is that our experiments are all conducted in the fully-supervised setting rather than few-shot setting.

NLU Tasks. First, we include datasets from SuperGLUE (Wang et al., 2019) to test P-tuning v2's general NLU ability. Additionally, we introduce a suite of sequence labeling tasks, including named entity recognition (Sang and De Meulder, 2003; Weischedel et al., 2013; Carreras and Màrquez, 2004), extractive Question Answering (Rajpurkar et al., 2016), and semantic role labeling (Carreras and Màrquez, 2005; Pradhan et al., 2012)).

Pre-trained Models. We include BERT-large (Devlin et al., 2018), RoBERTa-large (Liu et al., 2019), DeBERTa-xlarge (He et al., 2020), GLM-xlarge/xxlarge (Du et al., 2021) for evaluation. They are all bidirectional models designed for NLU tasks, covering a wide range of sizes from about 300M to 10B.

Multitask Learning. For the multi-task setting, we combine the training sets of the datasets in each task type (e.g., combing all training sets of semantic role labeling). We use separate linear classifiers for each dataset while sharing the continuous prompts (Cf. Appendix A).

	SST-2	RTE	BoolQ	CB
CLS & linear head	96.3	88.4	84.8	96.4
Verbalizer & LM head	95.8	86.6	84.6	94.6

Table 4: Comparison between [CLS] label with linear head and verbalizer with LM head on RoBERTa-large.

4.1 P-tuning v2: Across Scales

Table 2 presents P-tuning v2's performances across model scales. In SuperGLUE, performances of Lester et al. (2021) and P-tuning at smaller scales can be quite poor. On the contrary, P-tuning v2 matches the fine-tuning performance in all the tasks at a smaller scale. P-tuning v2 even significantly outperforms fine-tuning on RTE.

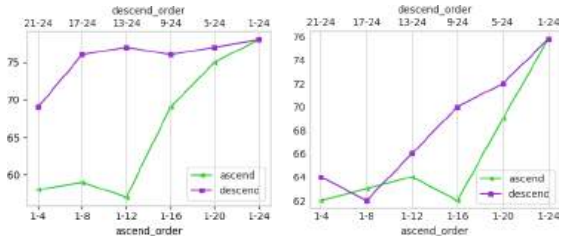
In terms of larger scales (2B to 10B) with GLM (Du et al., 2021), the gap between Lester et al. (2021); Liu et al. (2021) and fine-tuning is gradually narrowed down. On 10B scale, we have a similar observation as Lester et al. (2021) reports, that prompt tuning becomes competitive to fine-tuning. That said, P-tuning v2 is always comparable to fine-tuning at all scales but with only 0.1% task-specific parameters needed comparing to fine-tuning.

4.2 P-tuning v2: Across Tasks

From Table 3, we observe that P-tuning v2 can be generally comparable to fine-tuning on all tasks. P-tuning and Lester et al. (2021) show much poorer performance, especially on QA, which might be the most challenging of the three tasks. We also notice that there are some abnormal results of Lester et al. (2021) and P-tuning on SQuAD 2.0. This is probably because SQuAD 2.0 contains unanswerable questions, which causes optimization challenges for single-layer prompt tuning. Multi-task learning generally brings significant improvements to P-Tuning v2 over most tasks except for QA.

4.3 Ablation Study

Verbalizer with LM head v.s. [CLS] label with linear head. Verbalizer with LM head has been a central component in previous prompt tuning approaches. However, for P-tuning v2 in a supervised setting, it is affordable to tune a linear head with about several thousand parameters. We present our comparison in Table 4, where we keep other hyperparameters and only change [CLS] label with linear head to verbalizer with LM head. Here, for simplicity, we use "true" and "false" for SST-2, RTE and BoolQ; "true", "false" and "neutral" for CB. Results indicate that there is no significant difference between performances of verbalizer and [CLS].



(a) RTE (b) BoolQ

Figure 3: Ablation study on prompt depth using BERT-large. "[x-y]" refers to the layer-interval we add continuous prompts (e.g., "21-24" means we are add prompts to transformer layers from 21 to 24). Same amount of continuous prompts added to deeper transformer layers (i.e., more close to the output layer) can yield a better performance than those added to beginning layers.

Prompt depth. The main difference between Lester et al. (2021); (Liu et al., 2021) and P-tuning v2 is the multi-layer continuous prompts. To verify its exact influence, given a certain number of k layers to add prompts, we select them in both ascending and descending order to add prompts; for the rest layers, we left them untouched. As shown in Figure 3, with the same amount of parameters (i.e., num of transformer layers to add prompts), adding them in the descending order is always better than in the ascending order. In the RTE case, only adding prompts to layers 17-24 can yield a very close performance to all layers.

5 Conclusions

We present P-tuning v2, a prompt tuning method. Despite its relatively limited technical novelty, it contributes to a novel finding that prompt tuning can be comparable to fine-tuning universally across scales (from 330M to 10B parameters) and tasks. With high accuracy and parameter efficiency, P-Tuning v2 can be a potential alternative for fine-tuning and a strong baseline for future work.

ACKNOWLEDGEMENT

We would like to thank the anonymous reviewers for their suggestions and comments. Jie Tang is supported by the NSFC for Distinguished Young Scholar (61825602) and NSFC (61836013). Kaixuan Ji is supported by Tsinghua University Initiative Scientific Research Program and DCST Student Academic Training Program.

Prefix-Tuning: Optimizing Continuous Prompts for Generation

Xiang Lisa Li
Stanford University
xlisali@stanford.edu

Percy Liang
Stanford University
pliang@cs.stanford.edu

Abstract

Fine-tuning is the de facto way to leverage large pretrained language models to perform downstream tasks. However, it modifies all the language model parameters and therefore necessitates storing a full copy for each task. In this paper, we propose prefix-tuning, a lightweight alternative to fine-tuning for natural language generation tasks, which keeps language model parameters frozen, but optimizes a small *continuous task-specific* vector (called the prefix). Prefix-tuning draws inspiration from prompting, allowing subsequent tokens to attend to this prefix as if it were "virtual tokens". We apply prefix-tuning to GPT-2 for table-to-text generation and to BART for summarization. We find that by learning only 0.1% of the parameters, prefix-tuning obtains comparable performance in the full data setting, outperforms fine-tuning in low-data settings, and extrapolates better to examples with topics unseen during training.

1 Introduction

Fine-tuning is the prevalent paradigm for using large pretrained language models (LMs) (Radford et al., 2019; Devlin et al., 2019) to perform downstream tasks (e.g., summarization), but it requires updating and storing all the parameters of the LM. Consequently, to build and deploy NLP systems that rely on large pretrained LMs, one currently needs to store a modified copy of the LM parameters for each task. This can be prohibitively expensive, given the large size of current LMs; for example, GPT-2 has 774M parameters (Radford et al., 2019) and GPT-3 has 175B parameters (Brown et al., 2020).

A natural approach to this problem is *lightweight fine-tuning*, which freezes most of the pretrained parameters and augments the model with small trainable modules. For example, adapter-tuning

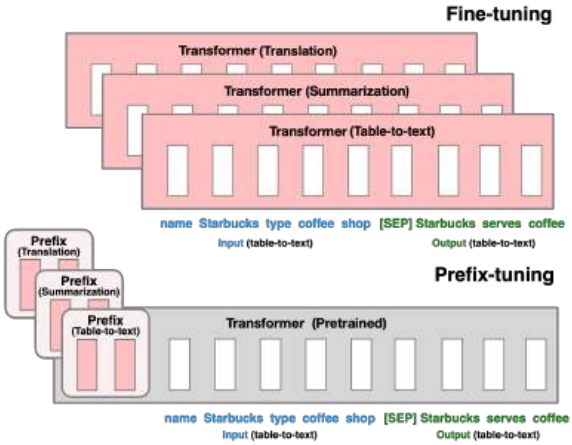


Figure 1: Fine-tuning (top) updates all Transformer parameters (the red Transformer box) and requires storing a full model copy for each task. We propose prefix-tuning (bottom), which freezes the Transformer parameters and only optimizes the prefix (the red prefix blocks). Consequently, we only need to store the prefix for each task, making prefix-tuning modular and space-efficient. Note that each vertical block denote transformer activations at one time step.

(Rebuffi et al., 2017; Houlsby et al., 2019) inserts additional task-specific layers between the layers of pretrained language models. Adapter-tuning has promising performance on natural language understanding and generation benchmarks, attaining comparable performance with fine-tuning while adding only around 2-4% task-specific parameters (Houlsby et al., 2019; Lin et al., 2020).

On the extreme end, GPT-3 (Brown et al., 2020) can be deployed without any task-specific tuning. Instead, users prepend a natural language task instruction (e.g., *TL;DR* for summarization) and a few examples to the task input; then generate the output from the LM. This approach is known as in-context learning or *prompting*.

In this paper, we propose *prefix-tuning*, a lightweight alternative to fine-tuning for natural language generation (NLG) tasks, inspired by prompting. Consider the task of generating a textual de-

scription of a data table, as shown in Figure 1, where the task input is a linearized table (e.g., "name: Starbucks | type: coffee shop") and the output is a textual description (e.g., "Starbucks serves coffee."). Prefix-tuning prepends a sequence of *continuous task-specific* vectors to the input, which we call a *prefix*, depicted by red blocks in Figure 1 (bottom). For subsequent tokens, the Transformer can attend to the prefix as if it were a sequence of "virtual tokens", but unlike prompting, the prefix consists entirely of free parameters which do not correspond to real tokens. In contrast to fine-tuning in Figure 1 (top), which updates all Transformer parameters and thus requires storing a tuned copy of the model for each task, prefix-tuning only optimizes the prefix. Consequently, we only need to store one copy of the large Transformer and a learned task-specific prefix, yielding a very small overhead for each additional task (e.g., 250K parameters for table-to-text).

In contrast to fine-tuning, prefix-tuning is modular: we train an upstream prefix which steers a downstream LM, which remains unmodified. Thus, a single LM can support many tasks at once. In the context of personalization where the tasks correspond to different users (Shokri and Shmatikov, 2015; McMahan et al., 2016), we could have a separate prefix for each user trained only on that user's data, thereby avoiding data cross-contamination. Moreover, the prefix-based architecture enables us to even process examples from multiple users/tasks in a single batch, something that is not possible with other lightweight fine-tuning approaches.

We evaluate prefix-tuning on table-to-text generation using GPT-2 and abstractive summarization using BART. In terms of storage, prefix-tuning stores 1000x fewer parameters than fine-tuning. In terms of performance when trained on full datasets, prefix-tuning and fine-tuning are comparable for table-to-text (§6.1), while prefix-tuning suffers a small degradation for summarization (§6.2). In low-data settings, prefix-tuning on average outperforms fine-tuning on both tasks (§6.3). Prefix-tuning also extrapolates better to tables (for table-to-text) and articles (for summarization) with unseen topics (§6.4).

2 Related Work

Fine-tuning for natural language generation. Current state-of-the-art systems for natural language generation are based on fine-tuning pretrained LMs. For table-to-text generation, Kale (2020) fine-tunes a sequence-to-sequence model (T5; Raffel et al., 2020). For extractive and abstractive summarization, researchers fine-tune masked language models (e.g., BERT; Devlin et al., 2019) and encode-decoder models (e.g., BART; Lewis et al., 2020) respectively (Zhong et al., 2020; Liu and Lapata, 2019; Raffel et al., 2020). For other conditional NLG tasks such as machine translation and dialogue generation, fine-tuning is also the prevalent paradigm (Zhang et al., 2020c; Stickland et al., 2020; Zhu et al., 2020; Liu et al., 2020). In this paper, we focus on table-to-text using GPT-2 and summarization using BART, but prefix-tuning can be applied to other generation tasks and pretrained models.

Lightweight fine-tuning. Lightweight fine-tuning freezes most of the pretrained parameters and modifies the pretrained model with small trainable modules. The key challenge is to identify high-performing architectures of the modules and the subset of pretrained parameters to tune. One line of research considers removing parameters: some model weights are ablated away by training a binary mask over model parameters (Zhao et al., 2020; Radiya-Dixit and Wang, 2020). Another line of research considers inserting parameters. For example, Zhang et al. (2020a) trains a "side" network that is fused with the pretrained model via summation; adapter-tuning inserts task-specific layers (adapters) between each layer of the pretrained LM (Houlsby et al., 2019; Lin et al., 2020; Rebuffi et al., 2017; Pfeiffer et al., 2020). Compared to this line of work, which tunes around 3.6% of the LM parameters, our method obtains a further 30x reduction in task-specific parameters, tuning only 0.1% while maintaining comparable performance.

Prompting. Prompting means prepending instructions and a few examples to the task input and generating the output from the LM. GPT-3 (Brown et al., 2020) uses manually designed prompts to adapt its generation for different tasks, and this framework is termed *in-context learning*. However, since Transformers can only condition on a bounded-length context (e.g., 2048 tokens for GPT-3), in-context learning is unable to fully exploit training sets longer than the context window. Sun and Lai (2020) also prompt by keywords to control for sentiment or topic of the generated sentence. In natural language understanding tasks, prompt

engineering has been explored in prior works for models like BERT and RoBERTa (Liu et al., 2019; Jiang et al., 2020; Schick and Schütze, 2020). For example, AutoPrompt (Shin et al., 2020) searches for a sequence of discrete trigger words and concatenates it with each input to elicit sentiment or factual knowledge from a masked LM. In contrast with AutoPrompt, our method optimizes continuous prefixes, which are more expressive (§7.2); moreover, we focus on language generation tasks.

Continuous vectors have been used to steer language models; for example, Subramani et al. (2020) showed that a pretrained LSTM language model can reconstruct arbitrary sentences by optimizing a continuous vector for each sentence, making the vector *input-specific*. In contrast, prefix-tuning optimizes a *task-specific* prefix that applies to all instances of that task. As a result, unlike the previous work whose application is limited to sentence reconstruction, prefix-tuning can be applied to NLG tasks.

Controllable generation. Controllable generation aims to steer a pretrained language model to match a sentence level attribute (e.g., positive sentiment or topic on sports). Such control can happen at training time: Keskar et al. (2019) pretrains the language model (CTRL) to condition on metadata such as keywords or URLs. Additionally, the control can happen at decoding time, by weighted decoding (GeDi, Krause et al., 2020) or iteratively updating the past activations (PPLM, Dathathri et al., 2020). However, there is no straightforward way to apply these controllable generation techniques to enforce fine-grained control over generated contents, as demanded by tasks like table-to-text and summarization.

3 Problem Statement

Consider a conditional generation task where the input is a context x and the output y is a sequence of tokens. We focus on two tasks, shown in Figure 2 (right): In table-to-text, x corresponds to a linearized data table and y is a textual description; in summarization, x is an article and y is a short summary.

3.1 Autoregressive LM

Assume we have an autoregressive language model $p_\phi(y \mid x)$ based on the Transformer (Vaswani et al., 2017) architecture (e.g., GPT-2; Radford et al., 2019) and parametrized by ϕ. As shown in Figure 2 (top), let $z = [x; y]$ be the concatenation of x and y; let $\mathsf{X}_{\mathsf{idx}}$ denote the sequence of indices that corresponds to x, and $\mathsf{Y}_{\mathsf{idx}}$ denote the same for y.

The activation at time step i is $h_i \in \mathbb{R}^d$, where $h_i = [h_i^{(1)}; \cdots; h_i^{(n)}]$ is a concatenation of all activation layers at this time step, and $h_i^{(j)}$ is the activation of the j-th Transformer layer at time step i.[1]

The autoregressive Transformer model computes h_i as a function of z_i and the past activations in its left context, as follows:

$$h_i = \mathrm{LM}_\phi(z_i, h_{<i}), \tag{1}$$

where the last layer of h_i is used to compute the distribution for the next token: $p_\phi(z_{i+1} \mid h_{\leq i}) = \mathrm{softmax}(W_\phi\, h_i^{(n)})$ and W_ϕ is a pretrained matrix that map $h_i^{(n)}$ to logits over the vocabulary.

3.2 Encoder-Decoder Architecture

We can also use an encoder-decoder architecture (e.g., BART; Lewis et al., 2020) to model $p_\phi(y \mid x)$, where x is encoded by the bidirectional encoder, and the decoder predicts y autoregressively (conditioned on the encoded x and its left context). We use the same indexing and activation notation, as shown in Figure 2 (bottom). h_i for all $i \in \mathsf{X}_{\mathsf{idx}}$ is computed by the bidirectional Transformer encoder; h_i for all $i \in \mathsf{Y}_{\mathsf{idx}}$ is computed by the autoregressive decoder using the same equation (1).

3.3 Method: Fine-tuning

In the fine-tuning framework, we initialize with the pretrained parameters ϕ. Here p_ϕ is a trainable language model distribution and we perform gradient updates on the following log-likelihood objective:

$$\max_\phi \; \log p_\phi(y \mid x) = \sum_{i \in \mathsf{Y}_{\mathsf{idx}}} \log p_\phi(z_i \mid h_{<i}). \tag{2}$$

4 Prefix-Tuning

We propose prefix-tuning as an alternative to fine-tuning for conditional generation tasks. We first provide intuition in §4.1 before defining our method formally in §4.2.

[1] $h_i^{(n)}$ is composed of a key-value pair. In GPT-2, the dimension of each key and value is 1024.

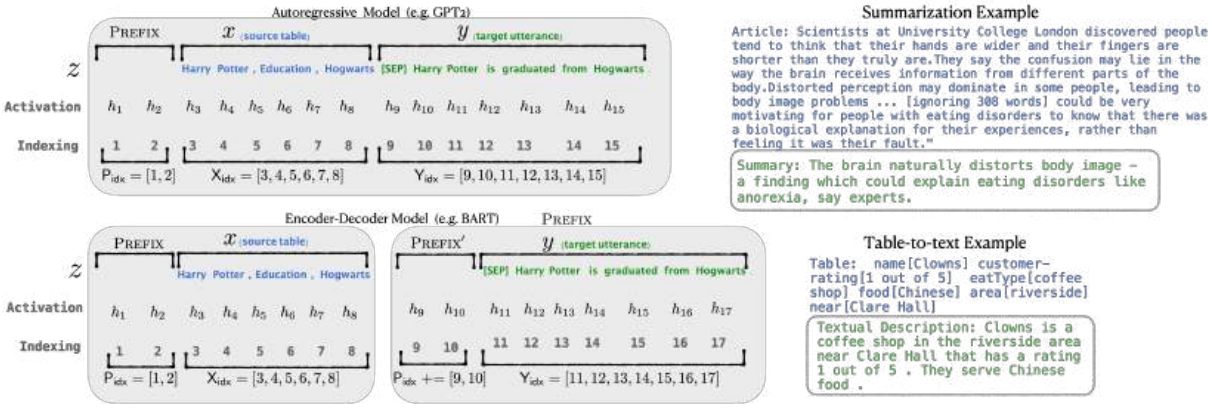


Figure 2: An annotated example of prefix-tuning using an autoregressive LM (top) and an encoder-decoder model (bottom). The prefix activations $\forall i \in \mathsf{P}_{\mathsf{idx}}, h_i$ are drawn from a trainable matrix P_θ. The remaining activations are computed by the Transformer.

4.1 Intuition

Based on intuition from prompting, we believe that having a proper context can steer the LM without changing its parameters. For example, if we want the LM to generate a word (e.g., Obama), we can prepend its common collocations as context (e.g., Barack), and the LM will assign much higher probability to the desired word. Extending this intuition beyond generating a single word or sentence, we want to find a context that steers the LM to solve an NLG task. Intuitively, the context can influence the encoding of x by guiding what to extract from x; and can influence the generation of y by steering the next token distribution. However, it's non-obvious whether such a context exists. Natural language task instructions (e.g., "summarize the following table in one sentence") might guide an expert annotator to solve the task, but fail for most pretrained LMs.[2] Data-driven optimization over the discrete instructions might help, but discrete optimization is computationally challenging.

Instead of optimizing over discrete tokens, we can optimize the instruction as continuous word embeddings, whose effects will be propagated upward to all Transformer activation layers and rightward to subsequent tokens. This is strictly more expressive than a discrete prompt which requires matching the embedding of a real word. Meanwhile, this is less expressive than intervening all layers of the activations (§7.2), which avoids long-range dependencies and includes more tunable parameters. Prefix-tuning, therefore, optimizes all layers of the prefix.

[2]In our preliminary experiments, GPT-2 and BART fail in this setting; the only exception is GPT-3.

4.2 Method

Prefix-tuning prepends a prefix for an autoregressive LM to obtain $z = [\textsc{Prefix}; x; y]$, or prepends prefixes for both encoder and encoder to obtain $z = [\textsc{Prefix}; x; \textsc{Prefix}'; y]$, as shown in Figure 2. Here, $\mathsf{P}_{\mathsf{idx}}$ denotes the sequence of prefix indices, and we use $|\mathsf{P}_{\mathsf{idx}}|$ to denote the length of the prefix.

We follow the recurrence relation in equation (1), except that the prefix are *free* parameters. Prefix-tuning initializes a trainable matrix P_θ (parametrized by θ) of dimension $|\mathsf{P}_{\mathsf{idx}}| \times \dim(h_i)$ to store the prefix parameters.

$$h_i = \begin{cases} P_\theta[i,:], & \text{if } i \in \mathsf{P}_{\mathsf{idx}}, \\ \mathrm{LM}_\phi(z_i, h_{<i}), & \text{otherwise.} \end{cases} \quad (3)$$

The training objective is the same as equation (2), but the set of trainable parameters changes: the language model parameters ϕ are fixed and the prefix parameters θ are the only trainable parameters.

Here, h_i (for all i) is a function of the trainable P_θ. When $i \in \mathsf{P}_{\mathsf{idx}}$, this is clear because h_i copies directly from P_θ. When $i \notin \mathsf{P}_{\mathsf{idx}}$, h_i still depends on P_θ, because the prefix activations are always in the left context and will therefore affect any activations to its right.

4.3 Parametrization of P_θ

Empirically, directly updating the P_θ parameters leads to unstable optimization and a slight drop in performance.[3] So we reparametrize the matrix $P_\theta[i,:] = \mathrm{MLP}_\theta(P'_\theta[i,:])$ by a smaller matrix (P'_θ) composed with a large feedforward neural network (MLP_θ). Note that P_θ and P'_θ has the same rows

[3]We find in preliminary experiments that directly optimizing the prefix is very sensitive to the learning rate and initialization.

dimension (i.e. the prefix length), but different columns dimension.[4] Once training is complete, these reparametrization parameters can be dropped, and only the prefix (P_θ) needs to be saved.

5 Experimental Setup

5.1 Datasets and Metrics

We evaluate on three standard neural generation datasets for the table-to-text task: E2E (Novikova et al., 2017), WebNLG (Gardent et al., 2017), and DART (Radev et al., 2020). The datasets are ordered by increasing complexity and size. E2E only has 1 domain (i.e. restaurant reviews); WebNLG has 14 domains, and DART is open-domain, using open-domain tables from Wikipedia.

The E2E dataset contains approximately 50K examples with 8 distinct fields; it contains multiple test references for one source table, and the average output length is 22.9. We use the official evaluation script, which reports BLEU (Papineni et al., 2002), NIST (Belz and Reiter, 2006), METEOR (Lavie and Agarwal, 2007), ROUGE-L (Lin, 2004), and CIDEr (Vedantam et al., 2015).

The WebNLG (Gardent et al., 2017) dataset consists of 22K examples, and the input x is a sequence of (subject, property, object) triples. The average output length is 22.5. In the training and validation splits, the input describes entities from 9 distinct DBpedia categories (e.g., Monument). The test split consists of two parts: the first half contains DB categories seen in training data, and the second half contains 5 unseen categories. These unseen categories are used to evaluate extrapolation. We use the official evaluation script, which reports BLEU, METEOR and TER (Snover et al., 2006).

DART (Radev et al., 2020) is an open domain table-to-text dataset, with similar input format (entity-relation-entity triples) as WebNLG. The average output length is 21.6. It consists of 82K examples from WikiSQL, WikiTableQuestions, E2E, and WebNLG and applies some manual or automated conversion. We use the official evaluation script and report BLEU, METEOR, TER, MoverScore (Zhao et al., 2019), BERTScore (Zhang et al., 2020b) and BLEURT (Sellam et al., 2020).

For the summarization task, we use the XSUM (Narayan et al., 2018) dataset, which is an abstractive summarization dataset on news articles. There are 225K examples. The average length of the articles is 431 words and the average length of the summaries is 23.3. We report ROUGE-1, ROUGE-2 and ROUGE-L.

5.2 Methods

For table-to-text generation, we compare prefix-tuning with three other methods: fine-tuning (FINE-TUNE), fine-tuning only the top 2 layers (FT-TOP2), and adapter-tuning (ADAPTER).[5] We also report the current state-of-the-art results on these datasets: On E2E, Shen et al. (2019) uses a pragmatically informed model without pretraining. On WebNLG, Kale (2020) fine-tunes T5-large. On DART, no official models trained on this dataset version are released.[6] For summarization, we compare against fine-tuning BART (Lewis et al., 2020).

5.3 Architectures and Hyperparameters

For table-to-text, we use GPT-2_{MEDIUM} and GPT-2_{LARGE}; the source tables are linearized.[7] For summarization, we use $\text{BART}_{\text{LARGE}}$,[8] and the source articles are truncated to 512 BPE tokens.

Our implementation is based on the Hugging Face Transformer models (Wolf et al., 2020). At training time, we use the AdamW optimizer (Loshchilov and Hutter, 2019) and a linear learning rate scheduler, as suggested by the Hugging Face default setup. The hyperparameters we tune include the number of epochs, batch size, learning rate, and prefix length. Hyperparameter details are in the appendix. A default setting trains for 10 epochs, using a batch size of 5, a learning rate of $5 \cdot 10^{-5}$ and a prefix length of 10. The table-to-text models are trained on TITAN Xp or GeForce GTX TITAN X machines. Prefix-tuning takes 0.2 hours per epochs to train on 22K examples , whereas fine-tuning takes around 0.3 hours. The summarization models are trained on Tesla V100 machines, taking 1.25h per epoch on the XSUM dataset.

At decoding time, for the three table-to-text datasets, we use beam search with a beam size of 5. For summarization, we use a beam size of 6

[4] P_θ has a dimension of $|\mathsf{P}_{\mathsf{idx}}| \times \dim(h_i)$ while P_θ has a dimension of $|\mathsf{P}_{\mathsf{idx}}| \times k$, where we choose $k = 512$ for table-to-text and 800 for summarization. MLP_θ maps from dimension k to $\dim(h_i)$

[5] Same implementation as Lin et al. (2020).

[6] The official benchmark model is trained on v.1.0.0 while the release dataset is v1.1.1.

[7] In comparison with natural language utterances, the linearized table is in an unnatural format, which might be challenging for pretrained LMs.

[8] We didn't include GPT-2 results for summarization because in our preliminary experiment, fine-tuning GPT-2 significantly underperforms fine-tuning BART on XSUM.

and length normalization of 0.8. Decoding takes 1.2 seconds per sentence (without batching) for table-to-text, and 2.6 seconds per batch (using a batch size of 10) for summarization.

6 Main Results

6.1 Table-to-text Generation

We find that adding only 0.1% task-specific parameters,[9] prefix-tuning is effective in table-to-text generation, outperforming other lightweight baselines (ADAPTER and FT-TOP2) and achieving a comparable performance with fine-tuning. This trend is true across all three datasets: E2E, WebNLG,[10] and DART.

For a fair comparison, we match the number of parameters for prefix-tuning and adapter-tuning to be 0.1%. Table 1 shows that prefix-tuning is significantly better than ADAPTER (0.1%), attaining 4.1 BLEU improvement per dataset on average. Even when we compare with fine-tuning (100%) and adapter-tuning (3.0%), which update significantly more parameters than prefix-tuning, prefix-tuning still achieves results comparable or better than those two systems. This demonstrates that prefix-tuning is more Pareto efficient than adapter-tuning, significantly reducing parameters while improving generation quality.

Additionally, attaining good performance on DART suggests that prefix-tuning can generalize to tables with diverse domains and a large pool of relations. We will delve deeper into extrapolation performance (i.e. generalization to unseen categories or topics) in §6.4.

Overall, prefix-tuning is an effective and space-efficient method to adapt GPT-2 to table-to-text generation. The learned prefix is expressive enough to steer GPT-2 in order to correctly extract contents from an unnatural format and generate a textual description. Prefix-tuning also scales well from GPT-2_{MEDIUM} to GPT-2_{LARGE}, suggesting it has the potential to scale to even larger models with a similar architecture, like GPT-3.

6.2 Summarization

As shown in Table 2, with 2% parameters, prefix-tuning obtains slightly lower performance than fine-tuning (36.05 vs. 37.25 in ROUGE-L). With only 0.1% parameters, prefix-tuning underperforms full fine-tuning (35.05 vs. 37.25). There are several differences between XSUM and the three table-to-text datasets which could account for why prefix-tuning has comparative advantage in table-to-text: (1) XSUM contains 4x more examples than the three table-to-text datasets on average; (2) the input articles are 17x longer than the linearized table input of table-to-text datasets on average; (3) summarization might be more complex than table-to-text because it requires reading comprehension and identifying key contents from an article.

6.3 Low-data Setting

Based on the results from table-to-text (§6.1) and summarization (§6.2), we observe that prefix-tuning has a comparative advantage when the number of training examples is smaller. To construct low-data settings, we subsample the full dataset (E2E for table-to-text and XSUM for summarization) to obtain small datasets of size $\{50, 100, 200, 500\}$. For each size, we sample 5 different datasets and average over 2 training random seeds. Thus, we average over 10 models to get an estimate for each low-data setting.[11]

Figure 3 (right) shows that prefix-tuning outperforms fine-tuning in low-data regimes by 2.9 BLEU on average, in addition to requiring many fewer parameters, but the gap narrows as the dataset size increases.

Qualitatively, Figure 3 (left) shows 8 examples generated by both prefix-tuning and fine-tuning models trained on different data levels. While both methods tend to undergenerate (missing table contents) in low data regimes, prefix-tuning tends to be more faithful than fine-tuning. For example, fine-tuning (100, 200)[12] falsely claims a low customer rating while the true rating is average, whereas prefix-tuning (100, 200) generates a description that is faithful to the table.

6.4 Extrapolation

We now investigate extrapolation performance to unseen topics for both table-to-text and summarization. In order to construct an extrapolation setting, we split the existing datasets so that training and test cover different topics. For table-to-text, the

[9] 250K for E2E, 250K for WebNLG, and 500K for DART vs. 345M GPT-2 parameters.

[10] The S,U,A columns in WebNLG represents SEEN, UNSEEN, and ALL respectively; SEEN categories appear at training time; UNSEEN categories only appears at test time; and ALL is the combination of the two.

[11] We also sample a dev split (with dev size = 30% $\times$ training size) for each training set. We use the dev split to choose hyperparameters and do early stopping.

[12] The number in the parenthesis refers to the training size.

	E2E					WebNLG									DART					
	BLEU	NIST	MET	R-L	CIDEr	BLEU			MET			TER ↓			BLEU	MET	TER ↓	Mover	BERT	BLEURT
						S	U	A	S	U	A	S	U	A						
GPT-2$_{\text{MEDIUM}}$																				
FINE-TUNE	68.2	8.62	**46.2**	71.0	2.47	**64.2**	27.7	46.5	**0.45**	0.30	0.38	**0.33**	0.76	0.53	46.2	**0.39**	**0.46**	**0.50**	**0.94**	**0.39**
FT-TOP2	68.1	8.59	46.0	70.8	2.41	53.6	18.9	36.0	0.38	0.23	0.31	0.49	0.99	0.72	41.0	0.34	0.56	0.43	0.93	0.21
ADAPTER(3%)	68.9	8.71	46.1	71.3	2.47	60.4	**48.3**	54.9	0.43	**0.38**	**0.41**	0.35	**0.45**	**0.39**	45.2	0.38	**0.46**	**0.50**	**0.94**	**0.39**
ADAPTER(0.1%)	66.3	8.41	45.0	69.8	2.40	54.5	45.1	50.2	0.39	0.36	0.38	0.40	0.46	0.43	42.4	0.36	0.48	0.47	**0.94**	0.33
PREFIX(0.1%)	**69.7**	**8.81**	46.1	**71.4**	**2.49**	62.9	45.6	**55.1**	0.44	**0.38**	**0.41**	0.35	0.49	0.41	**46.4**	0.38	**0.46**	**0.50**	**0.94**	**0.39**
GPT-2$_{\text{LARGE}}$																				
FINE-TUNE	68.5	8.78	46.0	69.9	2.45	**65.3**	43.1	55.5	**0.46**	0.38	**0.42**	**0.33**	0.53	0.42	**47.0**	**0.39**	0.46	**0.51**	**0.94**	**0.40**
Prefix	**70.3**	**8.85**	**46.2**	**71.7**	**2.47**	63.4	**47.7**	**56.3**	0.45	**0.39**	**0.42**	0.34	**0.48**	**0.40**	46.7	**0.39**	**0.45**	**0.51**	**0.94**	**0.40**
SOTA	68.6	8.70	45.3	70.8	2.37	63.9	52.8	57.1	0.46	0.41	0.44	-	-	-	-	-	-	-	-	-

Table 1: Metrics (higher is better, except for TER) for table-to-text generation on E2E (left), WebNLG (middle) and DART (right). With only 0.1% parameters, Prefix-tuning outperforms other lightweight baselines and achieves a comparable performance with fine-tuning. The best score is boldfaced for both GPT-2$_{\text{MEDIUM}}$ and GPT-2$_{\text{LARGE}}$.

Source	name : The Eagle | type : coffee shop | food : Chinese | price : cheap | customer rating : average | area : riverside | family friendly : no | near : Burger King
Prefix (50)	The Eagle is a cheap Chinese coffee shop located near Burger King.
Prefix (100)	The Eagle is a cheap coffee shop located in the riverside near Burger King. It has average customer ratings.
Prefix (200)	The Eagle is a cheap Chinese coffee shop located in the riverside area near Burger King. It has average customer ratings.
Prefix (500)	The Eagle is a coffee shop that serves Chinese food. It is located in the riverside area near Burger King. It has an average customer rating and is not family friendly.
FT (50)	The Eagle coffee shop is located in the riverside area near Burger King.
FT (100)	The Eagle is a cheap coffee shop near Burger King in the riverside area. It has a low customer rating and is not family friendly.
FT (200)	The Eagle is a cheap Chinese coffee shop with a low customer rating. It is located near Burger King in the riverside area.
FT (500)	The Eagle is a cheap Chinese coffee shop with average customer ratings. It is located in the riverside area near Burger King.

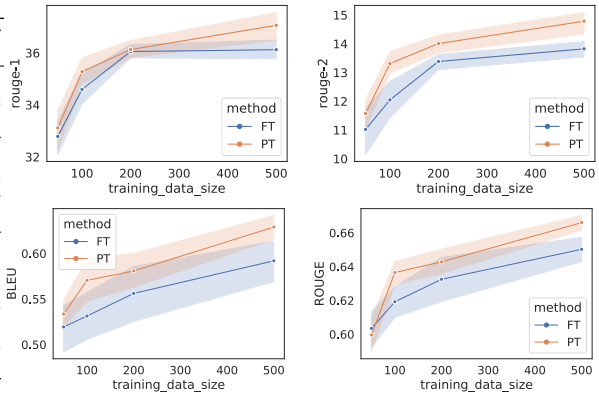


Figure 3: (Left) qualitative examples in lowdata settings. (Right) prefix-tuning (orange) outperforms fine-tuning (blue) in low-data regimes in addition to requiring many fewer parameters. The top two plots correspond to summarization, measured by ROUGE-1 and ROUGE-2. The bottom two plots correspond to table-to-text, measured by BLEU and ROUGE-L. The x-axis is the training size and the y-axis is the evaluation metric (higher is better).

	R-1 ↑	R-2 ↑	R-L ↑
FINE-TUNE(Lewis et al., 2020)	45.14	22.27	37.25
PREFIX(2%)	43.80	20.93	36.05
PREFIX(0.1%)	42.92	20.03	35.05

Table 2: Metrics for summarization on XSUM. Prefix-tuning slightly underperforms fine-tuning.

	news-to-sports			within-news		
	R-1 ↑	R-2 ↑	R-L ↑	R-1 ↑	R-2 ↑	R-L ↑
FINE-TUNE	38.15	15.51	30.26	39.20	16.35	31.15
PREFIX	39.23	16.74	31.51	39.41	16.87	31.47

Table 3: Extrapolation performance on XSUM. Prefix-tuning outperforms fine-tuning on both news-to-sports and within-news splits.

WebNLG dataset is labeled with table topics. There are 9 categories that appear in training and dev, denoted as SEEN and 5 categories that only appear at test time, denoted as UNSEEN. So we evaluate extrapolation by training on the SEEN categories and testing on the UNSEEN categories. For summarization, we construct two extrapolation data splits[13]: In `news-to-sports`, we train on news articles, and test on sports articles. In `within-news`, we train on {world, UK, business} news, and test on the remaining news categories (e.g., health, technology).

On both table-to-text and summarization, prefix-tuning has better extrapolation than fine-tuning under all metrics, as shown in Table 3 and the 'U' columns of Table 1 (middle).

We also find that adapter-tuning achieves good extrapolation performance, comparable with prefix-tuning, as shown in Table 1. This shared trend suggests that preserving LM parameters indeed has a positive impact on extrapolation. However, the

[13]XSUM dataset is drawn from BBC news, and we identify the topic of each article based on their URLs. Since "news" and "sports" are the two domains with the most articles, we create our first train/test split. Additionally, "news" has subdomains such as "UK", "world", and "technology". Consequently, we create a second data split, using the top 3 news subdomains as training data and the rest as test data.

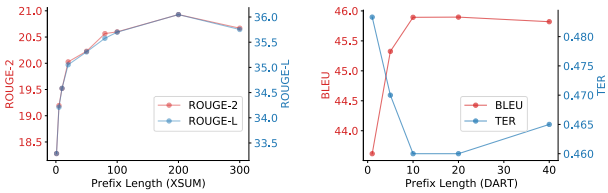


Figure 4: Prefix length vs. performance on summerization (left) and table-to-text (right). Performance increases as the prefix length increases up to a threshold (200 for summarization and 10 for table-to-text) and then a slight performance drop occurs. Each plot reports two metrics (on two vertical axes).

reason for such gains is an open question and we will discuss further in §8.

7 Intrinsic Evaluation

We compare different variants of prefix-tuning. §7.1 studies the impact of the prefix length. §7.2 studies tuning only the embedding layer, which is more akin to tuning a discrete prompt. §7.3 compares prefixing and infixing, which inserts trainable activations between x and y. §7.4 studies the impact of various prefix initialization strategies.

7.1 Prefix Length

A longer prefix means more trainable parameters, and therefore more expressive power. Figure 4 shows that performance increases as the prefix length increases up to a threshold (200 for summarization, 10 for table-to-text) and then a slight performance drop occurs.[14]

Empirically, longer prefixes have a negligible impact on inference speed, because attention computation over the entire prefix is parallellized on GPUs.

7.2 Full vs Embedding-only

Recall in §4.1, we discuss the option of optimizing the continuous embeddings of the "virtual tokens." We instantiate that idea and call it embedding-only ablation. The word embeddings are free parameters, and the upper activation layers are computed by the Transformer. Table 4 (top) shows that the performance drops significantly, suggesting that tuning only the embedding layer is not sufficiently expressive.

The embedding-only ablation upper bounds the performance of discrete prompt optimization (Shin et al., 2020), because discrete prompt restricts the embedding layer to exactly match the embedding of a real word. Consequently, we have this chain of increasing expressive power: discrete prompting $<$ embedding-only ablation $<$ prefix-tuning.

[14] Prefixes longer than the threshold lead to lower training loss, but slightly worse test performance, suggesting that they tend to overfit the training data.

	E2E BLEU	NIST	MET	ROUGE	CIDEr
PREFIX	69.7	8.81	46.1	71.4	2.49
Embedding-only: EMB-{PrefixLength}					
EMB-1	48.1	3.33	32.1	60.2	1.10
EMB-10	62.2	6.70	38.6	66.4	1.75
EMB-20	61.9	7.11	39.3	65.6	1.85
Infix-tuning: INFIX-{PrefixLength}					
INFIX-1	67.9	8.63	45.8	69.4	2.42
INFIX-10	67.2	8.48	45.8	69.9	2.40
INFIX-20	66.7	8.47	45.8	70.0	2.42

Table 4: Intrinsic evaluation of Embedding-only (§7.2) and Infixing (§7.3). Both Embedding-only ablation and Infix-tuning underperforms full prefix-tuning.

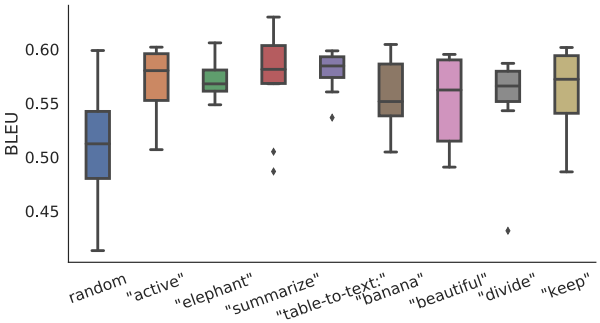


Figure 5: Initializing the prefix with activations of real words significantly outperforms random initialization, in low-data settings.

7.3 Prefixing vs Infixing

We also investigate how the trainable activations' position in the sequence affects performance. In prefix-tuning, we place them at the beginning $[\text{PREFIX}; x; y]$. We can also place the trainable activations between x and y (i.e. $[x; \text{INFIX}; y]$) and call this infix-tuning. Table 4 (bottom) shows that infix-tuning slightly underperforms prefix-tuning. We believe this is because prefix-tuning can affect the activations of x and y whereas infix-tuning can only influence the activations of y.

7.4 Initialization

We find that how the prefix is initialized has a large impact in low-data settings. Random initialization leads to low performance with high variance. Initializing the prefix with activations of real words

significantly improves generation, as shown in Figure 5. In particular, initializing with task relevant words such as "summarization" and "table-to-text" obtains slightly better performance than task irrelevant words such as "elephant" and "divide", but using real words is still better than random.

Since we initialize the prefix with activations of real words computed by the LM, this initialization strategy is concordant with preserving the pretrained LM as much as possible.

8 Discussion

In this section, we will discuss several favorable properties of prefix-tuning and some open problems.

8.1 Personalization

As we note in §1, prefix-tuning is advantageous when there are a large number of tasks that needs to be trained independently. One practical setting is user privacy (Shokri and Shmatikov, 2015; McMahan et al., 2016). In order to preserve user privacy, each user's data needs to be separated and a personalized model needs to be trained independently for each user. Consequently, each user can be regarded as an independent task. If there are millions of users, prefix-tuning can scale to this setting and maintain modularity, enabling flexible addition or deletion of users by adding or deleting their prefixes without cross-contamination.

8.2 Batching Across Users

Under the same personalization setting, prefix-tuning allows batching different users' queries even though they are backed by different prefixes. When multiple users query a cloud GPU device with their inputs, it is computationally efficient to put these users in the same batch. Prefix-tuning keeps the shared LM intact; consequently, batching requires a simple step of prepending the personalized prefix to user input, and all the remaining computation is unchanged. In contrast, we can't batch across different users in adapter-tuning, which has personalized adapters between shared Transformer layers.

8.3 Inductive Bias of Prefix-tuning

Recall that fine-tuning updates all pretrained parameters, whereas prefix-tuning and adapter-tuning preserve them. Since the language models are pretrained on general purpose corpus, preserving the LM parameters might help generalization to domains unseen during training. In concordance with this intuition, we observe that both prefix-tuning and adapter-tuning have significant performance gain in extrapolation settings (§6.4); however, the reason for such gain is an open question.

While prefix-tuning and adapter-tuning both freeze the pretrained parameters, they tune different sets of parameters to affect the activation layers of the Transformer. Recall that prefix-tuning keeps the LM intact and uses the prefix and the pretrained attention blocks to affect the subsequent activations; adapter-tuning inserts trainable modules between LM layers, which directly add residual vectors to the activations. Moreover, we observe that prefix-tuning requires vastly fewer parameters compared to adapter-tuning while maintaining comparable performance. We think this gain in parameter efficiency is because prefix-tuning keeps the pretrained LM intact as much as possible, and therefore exploits the LM more than adapter-tuning.

Concurrent work by Aghajanyan et al. (2020) uses intrinsic dimension to show that there exists a low dimension reparameterization that is as effective for fine-tuning as the full parameter space. This explains why good accuracy on downstream task can be obtained by updating only a small number of parameters. Our work echoes the finding by showing that good generation performance can be attained by updating a very small prefix.

9 Conclusion

We have proposed prefix-tuning, a lightweight alternative to fine-tuning that prepends a trainable continuous prefix for NLG tasks. We discover that despite learning 1000x fewer parameters than fine-tuning, prefix-tuning can maintain a comparable performance in a full data setting and outperforms fine-tuning in both low-data and extrapolation settings.

References

Armen Aghajanyan, Luke Zettlemoyer, and Sonal Gupta. 2020. Intrinsic dimensionality explains the effectiveness of language model fine-tuning.

Anja Belz and Ehud Reiter. 2006. Comparing automatic and human evaluation of NLG systems. In *11th Conference of the European Chapter of the Association for Computational Linguistics*, Trento, Italy. Association for Computational Linguistics.

Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda

Prefix-Tuning+: Modernizing Prefix-Tuning by Decoupling the Prefix from Attention

Haonan Wang[1,*] **Brian K. Chen**[1,2,*] **Siquan Li**[1,*] **Xinhe Liang**[1]
Hwee Kuan Lee[1,2,‡] **Kenji Kawaguchi**[1] **Tianyang Hu**[1,†]
[1]National University of Singapore [2]Bioinformatics Institute, A*STAR

Abstract

Parameter-Efficient Fine-Tuning (PEFT) methods have become crucial for rapidly adapting large language models (LLMs) to downstream tasks. Prefix-Tuning, an early and effective PEFT technique, demonstrated the ability to achieve performance comparable to full fine-tuning with significantly reduced computational and memory overhead. However, despite its earlier success, its effectiveness in training modern state-of-the-art LLMs has been very limited. In this work, we demonstrate empirically that Prefix-Tuning underperforms on LLMs because of an inherent tradeoff between input and prefix significance within the attention head. This motivates us to introduce Prefix-Tuning+, a novel architecture that generalizes the principles of Prefix-Tuning while addressing its shortcomings by shifting the prefix module out of the attention head itself. We further provide an overview of our construction process to guide future users when constructing their own context-based methods. Our experiments show that, across a diverse set of benchmarks, Prefix-Tuning+ consistently outperforms existing Prefix-Tuning methods. Notably, it achieves performance on par with the widely adopted LoRA method on several general benchmarks, highlighting the potential modern extension of Prefix-Tuning approaches. Our findings suggest that by overcoming its inherent limitations, Prefix-Tuning can remain a competitive and relevant research direction in the landscape of parameter-efficient LLM adaptation.

1 Introduction

Large Language Models (LLMs) have advanced at a remarkable pace within recent years, driven primarily by larger models and bigger training datasets [12, 28]. As a result, training and fine-tuning LLMs has become prohibitively expensive, with all but the biggest players unable to implement full parameter fine-tuning on state-of-the-art models. To remedy this, parameter-efficient fine-tuning (PEFT) methods have been introduced. One such approach is Prefix-Tuning (PT) [16], a technique which prepends trainable vectors to future inputs of each attention layer in the transformer. PT is extremely cheap to implement, while matching and even surpassing other bulkier methods in a variety of studies. However, as LLMs have grown to record sizes, PT has failed to perform well on the largest models, gradually losing popularity to other methods such as LoRA [11] and GaLore [40].

Earlier studies have primarily attributed this behaviour to PT's failure to reshape attention patterns within attention heads [27]. We show empirically that, while this applies to more shallow transformers, it does not extend to modern LLMs which tend to have a deep transformer architecture. Prefix-Tuning large language models can in fact result in a significant shift in the attention pattern. This leads to our

[*]Equal contribution.
[†]Correspondence to Tianyang Hu. Email: hutianyang.up@outlook.com.
[‡]Also affiliated with Nanyang Technological University, Singapore Eye Research Institute, Singapore International Research Laboratory on AI, and Singapore Institute for Clinical Sciences.

Preprint. Under review.

conclusion that an inability to alter attention patterns is not the reason behind PT's bad performance on state-of-the-art LLMs.

In this work, we argue that the real reason PT performs sub-optimally is its inherent tradeoff between prefix and input significance. When the prefix is long relative to input length, the model risks losing input specificity and being dominated by the prefix. When the input is long relative to prefix length, the impact of PT itself is greatly diminished. This tradeoff is a result of prefixes being included in the attention head itself. Motivated by this, we build on previous work [4] to propose Prefix-Tuning+ (PT+), which relocates the prefix outside the attention head and approximates it with an external module consisting of a trainable matrix and representation function. Diagnostic experiments suggest that PT+ is substantially more expressive than standard PT, reinforcing our choice of using the external module. We also provide a unified overview to the choices we make when extending PT to PT+, discussing how readers can potentially pick and choose what to keep when designing future context-based methods.

To evaluate the performance of PT+, we run extensive experiments in the few-shot data setting comparing it to other popular training methods such as LoRA and PT. Our experiments show that across multiple popular benchmarks, PT+ can compare directly with LoRA which is considered state-of-the-art. In a few cases it can even exceed it. Regular PT flounders in comparison.

Our work presents the following key contributions:

- We demonstrate empirically that Prefix-Tuning performs badly on modern LLMs because of an inherent tradeoff between input and prefix significance within the attention head.
- We introduce Prefix-Tuning+, a novel architecture based on Prefix-Tuning that isolates the prefix module outside of the attention head. We further provide a unified overview of our decision making process in constructing PT+ to guide users when constructing future context-based methods.
- We perform extensive experiments to show the efficacy of PT+. Our experiments show that, in the few-shot setting, PT+ is competitive with state-of-the-art approaches such as LoRA—achieving an average absolute improvement of **8.1%** over LoRA and **29.4%** over Prefix-Tuning across all six evaluated settings.

This serves as a proof of concept that, when the prefix information is isolated from the attention head like in PT+, prefix-tuning methods can serve as a viable alternative to current SOTA methods and is an exciting future area of research.

2 Related Work

Parameter-efficient fine-tuning (PEFT) [21, 8] adapts large language models (LLMs) by optimizing only a limited set of parameters while freezing the majority of pre-trained weights, reducing computational and memory demands. This approach enables rapid model adaptation to downstream tasks, facilitating deployment in resource-constrained environments without sacrificing performance [9].

Weight-Based PEFT Methods. LoRA [11] represents the most widely adopted weight-based PEFT method, introducing small, trainable low-rank matrices into transformer layers while freezing the original weight matrices. Variants such as QLoRA [7] and LoRA+ [10] refine this concept further, projecting the model's weights onto low-dimensional subspaces to achieve efficiency comparable to full fine-tuning at significantly reduced computational cost. However, these methods primarily adjust linear layers within transformer blocks, indirectly affecting internal attention patterns, and potentially limiting their flexibility in adapting attention patterns and behaviors explicitly.

Context-Based PEFT Methods. In contrast to weight-based methods, context-based PEFT methods directly alter the input context provided to LLMs without modifying the model's weights. Prominent examples include P-Tuning [18, 19], Prompt Tuning [15], and Prefix-Tuning [16]. Among these, Prefix-Tuning has been recognized for its exceptional parameter efficiency, achieving performance close to full fine-tuning on generation tasks. Nevertheless, Prefix-Tuning faces significant scalability issues, as performance quickly saturates or even declines with increasing prefix length [26, 27], thereby limiting its effectiveness in learning novel tasks that substantially differ from the pretraining distributions. Addressing these limitations is crucial for enhancing the versatility and applicability of context-based PEFT approaches. In this work, we present a unified view to better understand context-based PEFT methods and propose advancements that extend beyond traditional prefix-tuning.

3 Preliminaries

Transformer models were introduced to address sequence-to-sequence tasks and primarily consist of attention layers, feed forward networks, and other task specific modules [34]. In this paper, we assume inputs take the form $X = [x_1, \ldots, x_n]$, where each X is a sequence of tokens X_i where $X_i \in \mathbb{R}^d$ for all $i \in [n]$ such that $X \in \mathbb{R}^{n \times d}$.

3.1 The Attention Mechanism

Attention modules are a key component of transformers which accepts the entire sequence as an input. Typically, attention layers consist of multiple heads, each with a separate set of parameters. For notational simplicity we focus on single headed attention. A single attention head takes the form:

Definition 1 (Single-headed Attention) *Given input $X \in \mathbb{R}^{N \times d}$ and trainable matrices $W_Q, W_K \in \mathbb{R}^{d \times d_K}, W_V \in \mathbb{R}^{d \times d_V}$. A single attention head takes the form:*

$$O = \text{Attn}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_K}} + M\right)V, \tag{1}$$

where O is the output, $Q = XW_Q$, $K = XW_K$ and $V = XW_V$ and M is a causal mask. Based on [13], the attention head can be expressed as:

$$o_i^\top = \frac{\sum_{j \leq i} \text{sim}(q_i, k_j) v_j^\top}{\sum_{j \leq i} \text{sim}(q_i, k_j)}. \tag{2}$$

o_i is the i-th output token whilst $q_i = x_i W_Q$, $k_i = x_i W_K$ and $v_i = x_i W_V$ and $\text{sim}(q, k) = \exp(\frac{qk^\top}{\sqrt{d_K}})$ is a similarity score.

3.2 Prefix-Tuning

Prefix-tuning was initially motivated by the phenomenon of prompting and in-context learning (ICL):

Definition 2 (In-Context Learning) *ICL allows large language models to adapt to new tasks by prepending demonstration prompts to the input based on a specified criteria. Given context prompt $[x'_1, ..., x'_p]$ and input X, the new prompt becomes: $X^{ICL} = [x'_1, ..., x'_p, x_1, ..., x_n]$.*

Given the broad success of ICL, prefix tuning was introduced as a natural generalization of this principle. Rather than selecting tokens which correspond to elements available in the model's vocabulary, soft-tokens (i.e trainable vectors) are prepended to future model inputs:

Definition 3 (Prefix-Tuning) *Prefix-Tuning (PT) is a form of parameter-efficient fine-tuning that prepends a sequence of vectors to the inputs. Given prefix $[s_1, ..., s_p]$, where $s_i \in \mathbb{R}^d$ for all i, and input X, the new prompt becomes $X^{pt} = [s_1, ..., s_p, x_1, ..., x_n]$. The vectors $\{s_i\}_{i=1}^p$ are then trained based on traditional gradient based methods while the rest of the model weights are frozen.*

Referring to Equation (2), the inclusion of prefix $[s_1, ..., s_p]$ yields the following output:

$$o_i^{pt\,\top} = \frac{\sum_{j \leq i} \text{sim}(q_i, k_j) v_j^\top + \sum_{j \leq p} \text{sim}(q_i, W_K s_j)(W_V s_j)^\top}{\sum_{j \leq i} \text{sim}(q_i, k_j) + \sum_{j \leq p} \text{sim}(q_i, W_K s_j)}. \tag{3}$$

Any form of ICL is a special instance of prefix-tuning but *not* vice-versa, making prefix-tuning a more flexible and expressive form of fine-tuning compared with prompting and ICL methods.

Compared with full parameter fine-tuning and even most other PeFTs, prefix-tuning offers an extremely light-weight training approach. Research shows that prefix-tuning excels in low-data or few-shot settings and when guiding the model to leverage a mix of its pretrained tasks, rather than learning entirely new tasks from scratch.

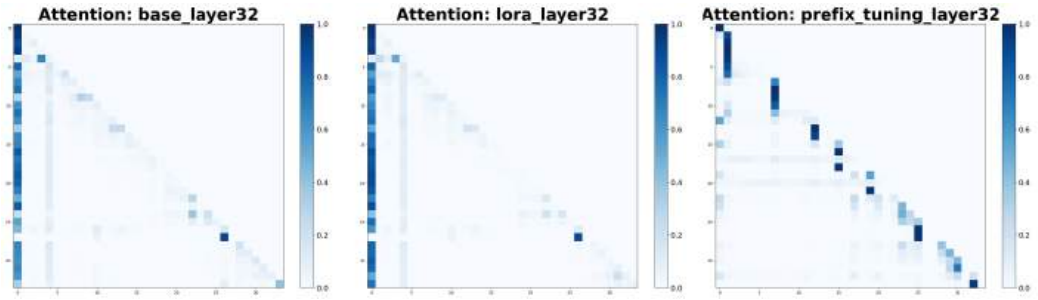


Figure 2: Attention Map of LLaMA2-7B-Chat, and its LoRA and Prefix-Tuning fine-tuned versions.

4 Limitations of prefix-tuning in LLMs

In the previous section, we noted that PT is particularly effective when leveraging pretrained tasks. With the continual increase in the size and capability of large language models (LLMs), supported by an expanding pretraining corpus, one might anticipate a corresponding rise in the prominence and effectiveness of PT. However, contrary to expectations, the adoption of prefix-tuning has significantly declined in recent years, as evidenced by its sparse implementation on state-of-the-art models available in repositories such as Hugging Face. This diminished popularity is primarily due to PT's underwhelming performance with larger and more complex models, which manifests in reduced accuracy and instability. As depicted in Figure 1, Prefix-Tuning consistently performs under compared to LoRA on three commonly used generative classification benchmarks, despite introducing a similar number of new parameters (see Section 6 for details). With the advent of LoRA, a Parameter-Efficient Fine-Tuning (PEFT) method that consistently outperforms Prefix-Tuning on established benchmarks—the overall relevance and applicability of Prefix-Tuning methods have been increasingly questioned.

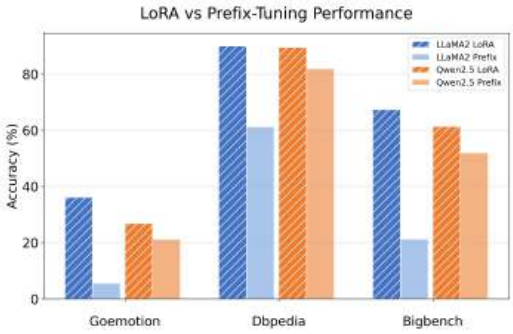


Figure 1: Performance comparison between Prefix-Tuning and LoRA.

4.1 Does Prefix-Tuning alter the attention pattern?

So why doesn't Prefix-Tuning behave well on state-of-the-art LLMs? The popular stance is that PT cannot alter the attention distribution in the attention heads [27]. Previous work [27] demonstrates that prefix-tuning is only capable of biasing the attention layer activations which forms a severe limitation. This is shown to be true for single-layer transformers and shallow transformers in general. In this study, we argue that, while this analysis is indicative for shallow transformers, it does not capture how PT behaves on LLMs, which are deep multi-layer transformers. Our experiments in Figure 2 show that PT can modify the attention pattern of LLMs significantly, despite having bad performance (experiment details are differed to Appendix B.2). This leads us to believe that an inability to affect the attention pattern is not why PT performs badly.

4.2 Tradeoff between prefix and input significance

In this section, we argue that the fundamental limitation of Prefix-Tuning is the inherent tradeoff between the significance of the prefix and the input. This can be observed by rewriting Equation (3) based on the work by [27] as follows:

$$o_i^{pt\,\top} = (1-\alpha_i)o_i^\top + \sum_{j\le p}\alpha_{ij}v_j'^\top, \tag{4}$$

where $\alpha_{ij} = \frac{\text{sim}(q_i, W_K s_j)}{\sum_{j\le i}\text{sim}(q_i,k_j)+\sum_{j\le p}\text{sim}(q_i,W_K s_j)}$, $\alpha_i = \sum_{j\le p}\alpha_{ij}$ and $v_j'^\top = W_V s_j'$.

Equation (4) shows that the output with prefix-tuning can be represented as a linear combination between the attention from the input o_i and the attention from each prefix v_j' with weights α_{ij}. Prefix-

Tuning mainly does two things: re-weights the original attention output and adds query-dependent bias vectors.

When the prefix is long relative to input length: In this case, we can expect the value of α to be large, which results in a greater change in the attention pattern since the base model's attention pattern is mainly dependent on o_i; this explains our observations in Figure 2. To further verify, we have conducted experiments with different prefix lengths and measured the attention pattern changes using the REEF framework [39]. Our results in Table 4 confirms that as prefix length increases, the deviation from the base attention pattern grows. Details can be found in Appendix B.3. What happens when you have a large α is a smaller contribution from the input itself. The model then has reduced specificity regarding each input and risks being dominated by the prefixes. Too little significance may be placed upon the input itself.

This is further exacerbated by the fact that, as the length of the prefix increases, prefix-tuning is unable to make full use of the space spanned by the vectors $\{W_V s_i\}_{i=1}^{p}$. This phenomenon is also noticed by [27] and is attributed to the competing optimization goals for the prefix s_i. The prefix both needs to grab attention through $W_K s_i$ and determine direction through $W_V s_i$.

When the input is long relative to prefix length: we can expect the value of α to be small. The opposite issue arises because when each α_i is small, the contribution of the prefix term is diminished. As LLMs get more and more capable, relying more on long sequences arising from techniques such as chain-of-thought reasoning [35], it is understandable for the effectiveness of prefix-tuning to be severely limited. Too little significance has been placed upon the prefix-tuning.

5 Prefix-Tuning+: Method and Framework

5.1 Motivation and Construction

A key insight from Section 4.2 is that the trade-off between prefix and input importance stems from the prefix's confinement within the attention head. This motivates Prefix-Tuning+, a novel extension of PT which seeks to bring the prefix information out of the attention head itself.

We first draw the terms containing the prefix information out of the attention head by splitting Equation (3) into:

$$o_i^{pt\top} = \lambda \frac{\sum_{j\le i} \mathrm{sim}(q_i, k_j) v_j^\top}{\sum_{j\le i} \mathrm{sim}(q_i, k_j)} + (1-\lambda) \frac{\sum_{j\le p} \mathrm{sim}(q_i, W_K s_j)(W_V s_j)^\top}{\sum_{j\le p} \mathrm{sim}(q_i, W_K s_j)}, \tag{5}$$

where $\lambda \in [0, 1]$ is a constant. This replaces the softmax regularization tradeoff, which is dependent on the length of the input and context, with a fixed convex linear combination similar to previous works [24, 36]. Then, we approximate the similarity metric $\mathrm{sim}(\cdot,\cdot)$ with a kernel feature map ϕ such that $\mathrm{sim}(\cdot,\cdot) \approx \phi(\cdot)^\top \phi(\cdot)$. We have

$$o_i^{pt\top} = \lambda \frac{\sum_{j\le i} \mathrm{sim}(q_i, k_j) v_j^\top}{\sum_{j\le i} \mathrm{sim}(q_i, k_j)} + (1-\lambda) \frac{\phi(q_i)^\top \sum_{j\le p} \phi(W_K s_j)(W_V s_j)^\top}{\phi(q_i)^\top \sum_{j\le p} \phi(W_K s_j)}. \tag{6}$$

A similar approach is used in [4] to approximate in-context learning prompts, which has shown that the bias term $b_1 = \sum_{j\le p} \phi(W_K s_j)(W_V s_j)^\top$ is capable of capturing contextual prompt or prefix information. The natural generalization of this step is to replace the bias b_1 by a more expressive, trainable matrix M, and the analogous term $b_2 = \sum_{j\le p} \phi(W_K s_j)$ by a trainable matrix N, which yields:

$$o_i^{pt\top} = \lambda \frac{\sum_{j\le i} \mathrm{sim}(q_i, k_j) v_j^\top}{\sum_{j\le i} \mathrm{sim}(q_i, k_j)} + (1-\lambda) \frac{\phi(q_i)^\top M}{\phi(q_i)^\top N}. \tag{7}$$

In practice, we make two more modifications during application. First, due to the trainable nature of M and layer normalization, λ can be absorbed into the trainable weights and is not necessary. Second, $\phi(q_i)^\top N$ is no longer meaningful for regularization, so we remove it. Therefore, the final attention output of the Prefix-Tuning+ architecture has the following form:

$$o_i^{pt+\top} = \frac{\sum_{j\le i} \mathrm{sim}(q_i, k_j) v_j^\top}{\sum_{j\le i} \mathrm{sim}(q_i, k_j)} + \phi(q_i)^\top M. \tag{8}$$

Choice of feature map. Regarding the choice of ϕ there are several viable options which represent a tradeoff between expressivity and cost. A few from existing literature include $\phi(x) = \text{elu}(x)$ [13] and $\phi_W(x) = \text{ReLU}(Wx + b)$ [23]. In this study, as a proof of concept, we conduct all experiments with $\phi(x) = \text{elu}(x)$. This is because it is the easiest to implement and offers a good proof of concept regarding the viability of our approach. Other choices may offer more expressiveness and better performance but would require significantly more detailed tuning so we leave it to future work. Further details on construction of the Prefix-Tuning+ modules are offered in the appendix.

Remark 1 (Expressiveness) *By choosing $\phi_W(x) = \text{ReLU}(Wx + b)$, the term $\phi_W(q_i)M$ becomes effectively a single-layer MLP. Depending on future choices for $\phi(\cdot)$, Prefix-Tuning+ has the ability to be extremely expressive, matching methods such as full fine-tuning and LoRA.*

5.2 A Unified View for Context Based Methods

This section outlines the design choices and intermediate stages behind PT and PT+ in general, offering the rationale for each to guide future implementation decisions. We first refer to Equation (4). The most elementary version of prefix-tuning is ICL, where each vector v'_j corresponds to an input-vocabulary token prepended to the input of the transformer. Based on a decision to increase expressivity with the added need for training, we have prompt-tuning, where v'_j are replaced with trainable soft prompts. Last but not least, there is the decision to improve expressivity with added computational and memory cost. This leads to PT which prepends these soft prompts to the inputs of the individual attention heads of the transformer. To arrive at PT+, there are two following decisions to be made:

1. Shift the prefix module out of the attention head
2. Approximate $\sum_{j \leq p} \text{sim}(., W_K s_j)$ by $\phi(\cdot)^\top M$

Choice 1: Shifting the prefix module out of the attention head is to avoid the limitations highlighted in Section 4.2. By doing so we avoid the α scaling on both the input and prefixes so there is no longer the same tradeoff between input contribution and prefix significance/contribution.

Choice 2: Replacing the original similarity metric by $\phi(\cdot)^\top M$ shifts the output from Equation (6) to Equation (7). By doing so, we lose some of the inherent structure of the attention mechanism. In return, we have an increase in model expressivity from the flexibility of a training matrix M. Since both PT and PT+ can be viewed as adding query-dependent d-dimensional bias terms to the transformer, we calculate the covariate output matrices of the bias from each and find the respective eigenvalue decay. From Figure 3, we see that with PT+, the top eigenvalues corresponding to the main principle components are large and decay slowly compared to PT. This indicates that the output bias spans many principal components rather than collapsing onto a handful of axes. In other words, Prefix-Tuning+ adds a bias from a more diverse, high-dimensional subspace. This is an intuitive proxy which indicates higher expressivity.

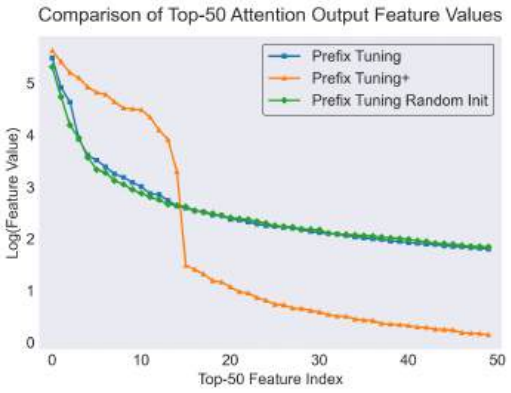


Figure 3: Spectrum of prefix representations.

In Prefix-Tuning+, both choices are used in conjunction. This does not have to be the case. Users can choose to keep the prefix term within the attention head and only apply choice 2. The resulting output is expressed as:

$$o_i^{pt\ \top} = \frac{\sum_{j \leq i} \text{sim}(q_i, k_j) v_j^\top + \phi(q_i)^\top M}{\sum_{j \leq i} \text{sim}(q_i, k_j) + \phi(q_i)^\top N}. \tag{9}$$

The opposite is also true, and prefix information can be brought out of the attention head through another module. In this work we choose to combine the two because we consider it expressive, easy

Table 1: Fine-Tuning Method Performance Comparison (Accuracy %). Results across datasets and models; best-performing results are in boldface, highlighting the effectiveness of Prefix-Tuning+.

Dataset	LLaMA2-7B-Chat				Qwen2.5-3B-Instruct			
	Prefix-Tuning+	Full	LoRA	Prefix	Prefix-Tuning+	Full	LoRA	Prefix
Goemotion	**45.2**	32.7	36.2	5.6	37.3	**37.8**	26.8	21.2
Dbpedia	**92.7**	92.6	90.1	61.3	**96.9**	94.4	89.5	82.0
Bigbench	**71.2**	38.8	67.4	21.3	**76.6**	67.4	61.4	52.0

to implement and a good proof of concept. However, in future research, what choices to implement for the optimal architecture is an interesting direction.

Remark 2 (The Memory Perspective) *We can view our method as explicitly treating the learnable matrix M as an internal memory store. Traditional context-based PEFTs, such as Prefix-Tuning, incorporate context memory by extending the KV inputs, tying the memory capacity to the prefix length. By linearizing attention and summing over the KV circuit, our approach decouples the memory capacity from sequence length and instead makes it proportional to the dimensionality of M, enabling more flexible storage of attention patterns. In practice, M allows the model to record and retrieve token interactions without altering the core attention weights by acting as an external memory module. This memory interface is both more direct and more parameter-efficient than auxiliary MLP-based memory modules, which typically require deep architectural changes, incurring higher costs.*

6 Experiments

In this section, we evaluate Prefix-Tuning+ across diverse tasks, models, and training settings, focusing on rapid adaptation, IID accuracy, and OOD generalization. We also investigate the impact of attention mechanisms and extend evaluations to practical alignment scenarios.

6.1 Experimental Setup

Datasets. We evaluate on four generative QA tasks:BigBench [31, 32], GoEmotions [6], DBpedia [14] and Banking77 [3]. We leave the detailed description of those dataset in Appendix A.1.

Training and Evaluation Protocol. We assess each method's ability to quickly adapt to downstream tasks in a few-shot setting by fine-tuning on up to five independent rounds of minimal data. In each round, we randomly sample one example per class (6 examples for BIG-bench, 28 for GoEmotions, and 14 for DBpedia) to form the entire training set. After fine-tuning, we report in-distribution (IID) accuracy on each dataset's standard test split, averaging results over the five rounds to mitigate sampling variability. Since the ability to quickly adapt to new tasks often comes at the cost of generalization, we also evaluate out-of-distribution (OOD) performance using the Banking77 intent-classification dataset without additional fine-tuning. During inference, models receive a multiple-choice prompt listing all 77 Banking77 intents and must select the most appropriate label for each query. OOD accuracy is computed as the proportion of test queries correctly classified, measuring how effectively learned features generalize to unseen domains. We perform this evaluation independently for each of the five models fine-tuned on different source datasets.

Models and Training Configuration. We experiment with two pre-trained language models to assess architectural effects: LLaMA2-7B-Chat and Qwen2.5-3B-Instruct. The LLaMA2 series models employ the multi-head attention (MHA) [34] and Qwen2.5 use grouped-query attention (GQA) [1]. GQA ties together query heads by sharing key/value projections, offering faster inference and lower memory usage, which allows us to examine if such architectural differences impact adaptation efficacy. Both models are used in their instruction version in order to test the OOD performance. We fine-tune these models using the AdamW [20] optimizer with a small learning rate and a fixed number of training steps (4000 steps). All methods use same small batch size (batch size 2).

Baselines. We compare Prefix-Tuning+ against several baseline approaches for adapting large language models, covering both parameter-efficient and traditional full fine-tuning, as well as a training-free prompt-based baseline:

- **Full Fine-Tuning**: All model parameters are fine-tuned on the minimal training set for each round. This represents the conventional approach where all weights of models are updated.

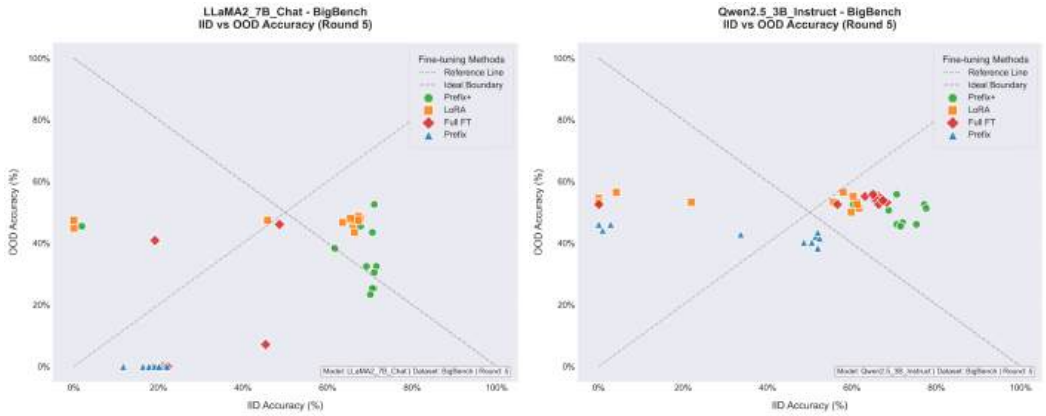


Figure 4: Pareto plots illustrating the trade-off between IID performance (on Bigbench) and OOD performance (on Banking77) for checkpoints of LLaMA2 and Qwen2.5 during training.

- **Low-rank adaptation (LoRA [11])**: LoRA freezes original model parameters and introduces trainable low-rank update matrices into each Transformer layer. Only these small rank-r matrices are learned, substantially reducing the number of trainable parameters. We set $r = 64$ to approximately match the parameter count introduced by Prefix-Tuning+.
- **Prefix-Tuning (PT [16])**: Standard prefix-tuning keeps all model weights fixed, learning only a continuous prefix vector that is prepended to the input at each Transformer layer. We follow the original implementation and set the prefix length $m = 32$.
- **In-Context Learning (ICL [2])**: Unlike the previous methods, ICL involves no parameter updates. Instead, the training examples are directly provided as demonstrations in the context at inference.

6.2 Supervised Fine-Tuning Performance Across Tasks

PEFT techniques aim to rapidly adapt large pre-trained language models (LLMs) to downstream tasks by updating a limited number of parameters. To study the effectiveness and adaptability of our proposed Prefix-Tuning+ across diverse classification scenarios, we conduct experiments on several tasks with the five round data setting. We summarize the accuracy results in Table 1, comparing Prefix-Tuning+ with various baseline approaches across the evaluated datasets. Our Prefix-Tuning+ consistently demonstrates superior or highly competitive performance compared to all baseline methods. Specifically, on BIG-bench, Prefix-Tuning+ achieves an accuracy of 71.2% with LLaMA2-7B-Chat and 76.6% with Qwen2.5-3B-Instruct, significantly outperforming LoRA, Prefix-Tuning, and full fine-tuning. On DBpedia, Prefix-Tuning+ also achieves top results (92.7% for LLaMA2, 96.9% for Qwen2.5), matching or exceeding the performance of the strongest baselines. For GoEmotions, Prefix-Tuning+ remains robust, reaching 45.2% accuracy with LLaMA2-7B-Chat and achieving a competitive 37.3% with Qwen2.5-3B-Instruct. These outcomes underscore Prefix-Tuning+ 's capability to effectively generalize and perform across varied classification tasks and model architectures.

6.3 Balancing In-Distribution Accuracy and Out-of-Distribution Generalization

An inherent IID-OOD performance trade-off typically emerges when models are trained to optimize for specific downstream tasks. In this section, we aims to study robustness of various fine-tuning approaches in effectively balancing IID performance with OOD resilience. Specifically, we examine the performance of the LLaMA2-7B-Chat and Qwen2.5-3B-Instruct models trained on the three datasets (BigBench, GoEmotions, and DBpedia). IID performance is measured directly on the hold-out part of those datasets, while OOD performance is evaluated using the Banking77 dataset. To provide a clear visualization, we present Pareto plots that depict the trade-off between IID (x-axis) and OOD (y-axis) performance. Each point on these plots represents the performance throughout training (from various checkpoints saved in different steps), with points of the same color corresponding to checkpoints from the same fine-tuning approach. The results of two models on BigBench are shown in Figure 4. These plots clearly demonstrate the performance trade-offs and highlight the differences in how each model generalizes from IID conditions to OOD scenarios. Notably, our proposed method consistently appears on the Pareto front, indicating that it achieves an optimal balance between IID and OOD performance. We leave results on more datasets in Appendix A.2.

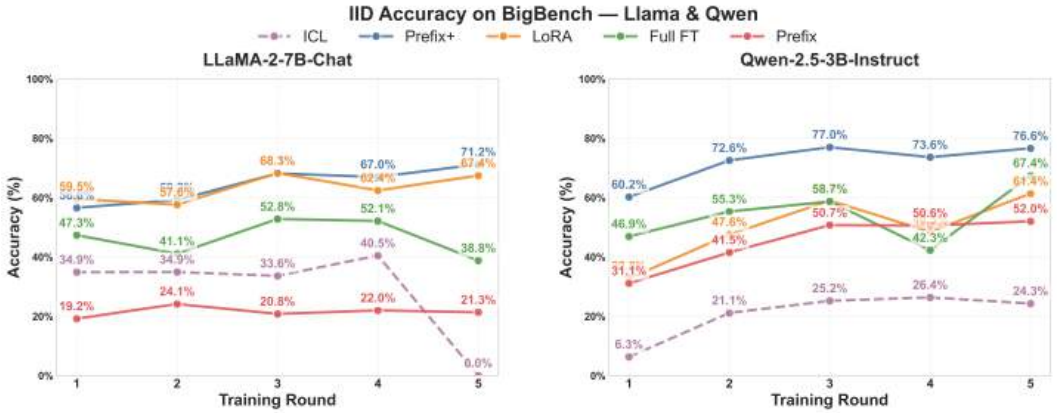


Figure 5: Performance over five incremental rounds of training data on BigBench. Prefix-Tuning+ consistently matches or exceeds baselines, with the largest gains observed on Qwen-2.5-3B-Instruct.

6.4 Performance Across Varying Data Sizes and Attention Mechanisms

To evaluate how effectively Prefix-Tuning+ scales with training set size and different attention mechanisms, we conducted experiments using the BigBench dataset, incrementally increasing dataset size over five rounds. We fine-tuned two distinct models, LLaMA-2-7B-Chat with standard attention and Qwen2.5-3B-Instruct with grouped-query attention (GQA)—using Prefix-Tuning+, Prefix-Tuning, LoRA, and full-parameter fine-tuning. Figure 5 illustrates the average performance across these rounds. Our analysis highlights two points: first, Prefix-Tuning+ maintains strong and consistent performance across different data scales and attention mechanisms, effectively matching or surpassing all baseline methods. Second, Prefix-Tuning+ shows particularly notable improvements when combined with GQA, outperforming both LoRA and full-parameter fine-tuning. These results indicates that Prefix-Tuning+ is effective when paired with the widely adopted grouped-query attention (GQA) mechanism, yielding superior performance compared to existing approaches. Additional results can be found in Appendix A.3.

6.5 Practical Alignment Tasks across Larger Datasets and Diverse Optimization Objectives

To study the effectiveness of our proposed Prefix-Tuning+ beyond generative text classification, we performed experiments aimed at aligning large language models (LLMs) more closely with human values and intentions. Specifically, we evaluated how well Prefix-Tuning+ performs when integrated with different preference optimization strategies. We employed the Qwen2.5-3B model optimized with Prefix-Tuning+ and compared its performance against LoRA, using three different training approaches: supervised fine-tuning (SFT)[25] on the Magpie-Ultra v0.1 dataset[37], and two preference-based methods—Direct Preference Optimization (DPO)[29] and Simple Preference Optimization (SimPO)[22]—using the binarized UltraFeedback dataset [5]. For each training method, we used a consistent dataset size of 10,000 samples to ensure fairness and comparability of results. Following training, we evaluated the models using AlpacaEval 2 [17], a standardized benchmark for alignment tasks. All experiments were implemented and executed using the LLaMAFactory framework [41]. Table 2 summarizes the improvement in win-rates achieved by each method. Prefix-Tuning+ consistently delivered higher win-rate increases compared to LoRA across all training objectives, highlighting its robustness and versatility. The advantage of Prefix-Tuning+ was particularly pronounced in preference-based settings (DPO and SimPO), where it notably outperformed LoRA. Interestingly, our experiments revealed a slight but consistent advantage of DPO over SimPO, contrary to prior findings [22]. We hypothesize that SimPO's comparatively weaker performance in our setup may stem from its sensitivity to hyperparameter configurations [30].

Table 2: Performance improvements of Prefix-Tuning+ over LoRA on alignment tasks using SFT, DPO, and SimPO objectives (evaluated with AlpacaEval 2).

Method	SFT	DPO	SimPO
LoRA	+0.49	+3.52	+1.24
Prefix-Tuning+	**+0.76**	**+4.66**	**+1.74**

7 Discussion

To conclude, in this work we argue that the reason why Prefix-Tuning has been ineffective when applied to modern large language models is that prefixes are "trapped" within the attention head. To remedy this, we introduce a novel architecture that generalizes upon existing Prefix-Tuning methods by approximating the prefix module and shifting it out of the attention head. Surprisingly, even with this slightly naive implementation, our model is able to match state-of-the-art methods such as LoRA on popular benchmarks in a few-shot setting, far outpacing previous prefix-tuning methods. We treat this as proof of concept that, if approached correctly, Prefix-Tuning methods can be competitive and are an exciting future avenue of research.

We also acknowledge the existing limitations of our work. Rather than presenting a clear alternative to existing PEFTs, Prefix-Tuning+ is primarily a proof of concept. The design of our method has yet to be thoroughly ablated. For instance, this line of work can potentially be improved utilizing a more powerful choice of feature map ϕ such as a learnable one. Further studies are needed to test the limits of our method in more tasks and with more training objectives.

Quiet-STaR: Language Models Can Teach Themselves to Think Before Speaking

Eric Zelikman
Stanford University

Georges Harik
Notbad AI Inc

Yijia Shao
Stanford University

Varuna Jayasiri
Notbad AI Inc

Nick Haber
Stanford University

Noah D. Goodman
Stanford University

Abstract

When writing and talking, people sometimes pause to think. Although reasoning-focused works have often framed reasoning as a method of answering questions or completing agentic tasks, reasoning is implicit in almost all written text. For example, this applies to the steps not stated between the lines of a proof or to the theory of mind underlying a conversation. In the Self-Taught Reasoner (STaR, Zelikman et al. 2022), useful thinking is learned by inferring rationales from few-shot examples in question-answering and learning from those that lead to a correct answer. This is a highly constrained setting – ideally, a language model could instead learn to infer unstated rationales in arbitrary text. We present **Quiet-STaR**, a generalization of STaR in which LMs learn to generate rationales at each token to explain future text, improving their predictions. We address key challenges, including 1) the computational cost of generating continuations, 2) the fact that the LM does not initially know how to generate or use internal thoughts, and 3) the need to predict beyond individual next tokens. To resolve these, we propose a tokenwise parallel sampling algorithm, using learnable tokens indicating a thought's start and end, and an extended teacher-forcing technique. Encouragingly, generated rationales disproportionately help model difficult-to-predict tokens and improve the LM's ability to directly answer difficult questions. In particular, after continued pretraining of an LM on a corpus of internet text with Quiet-STaR, we find zero-shot improvements on GSM8K (5.9%→10.9%) and CommonsenseQA (36.3%→47.2%) and observe a perplexity improvement of difficult tokens in natural text. Crucially, these improvements require no fine-tuning on these tasks. Quiet-STaR marks a step towards LMs that can learn to reason in a more general and scalable way.

"Life can only be understood backwards; but it must be lived forwards."

— Sren Kierkegaard

1 Introduction

Much of the meaning of text is hidden between the lines: without understanding why statements appear in a document, a reader has only a shallow understanding. Moreover, this has been repeatedly shown to be true for LMs as well, in the contexts of tasks ranging from commonsense reasoning to theorem proving to programming (Wei et al., 2022b; Nye et al., 2021; Zelikman et al., 2022; 2023a; Kojima et al., 2022). Reasoning about implications of text to predict later text has consistently been shown to improve LM performance on a variety of tasks, but methods for allowing LMs to learn from their reasoning (e.g., Zelikman et al. 2022) have focused on solving individual tasks or predefined sets of tasks (e.g., Wei et al. 2021b). These works rely on carefully curated datasets to provide either specific reasoning tasks or in some cases, the reasoning itself. We instead ask, if reasoning is implicit in all text, why shouldn't we leverage the task of language modeling to teach reasoning?

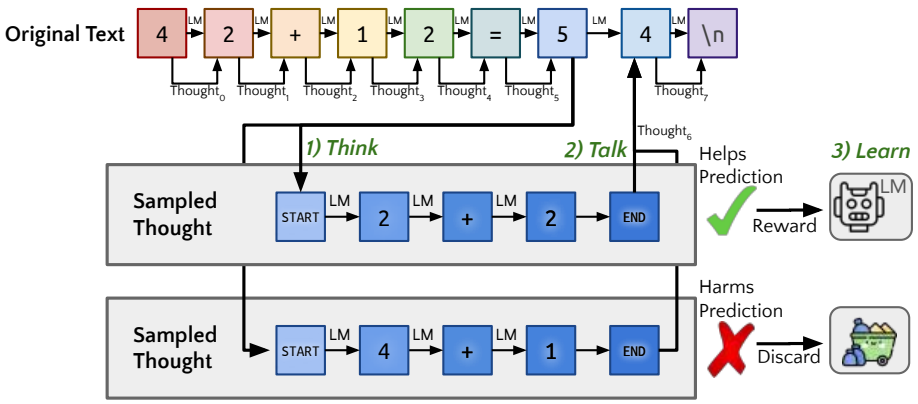


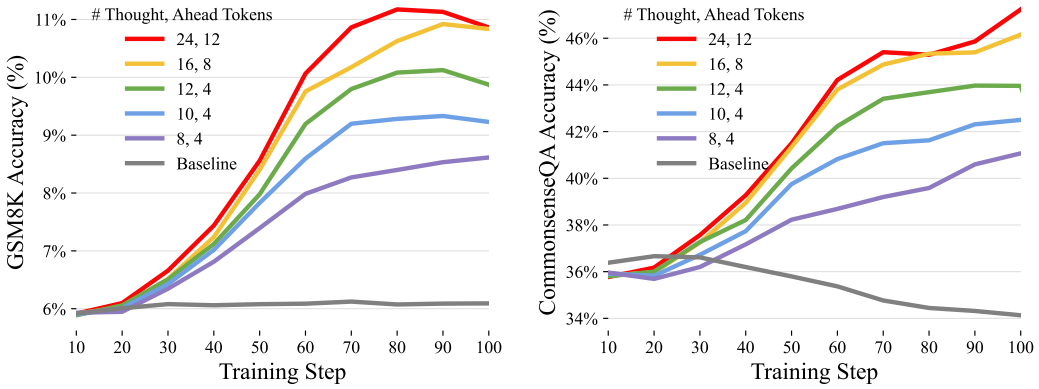
Figure 1: **Quiet-STaR**. We visualize the algorithm as applied during training to a single thought. We generate thoughts, in parallel, following all tokens in the text (think). The model produces a mixture of its next-token predictions with and without a thought (talk). We apply REINFORCE, as in STaR, to increase the likelihood of thoughts that help the model predict future text while discarding thoughts that make the future text less likely (learn).

In particular, the Self-Taught Reasoner (STaR, Zelikman et al. 2022) showed that LMs can bootstrap their reasoning ability on question-answering (QA) datasets by sampling rationales to attempt to answer questions, training on rationales if they led to a correct final answer, and then repeating this to iteratively solve more difficult problems. Yet, training from curated QA datasets limits the scale and generalizability of the rationales. QA datasets, especially high-quality ones, require thoughtful curation and will inherently only ever cover a subset of reasoning tasks. Thus, we extend STaR – instead of the LM learning to reason on particular tasks like mathematical QA, we train an LM to generate reasoning that helps it infer future text from a large internet text corpus. As a result, we allow the LM to learn from the diverse tasks present in language (Weber et al., 2021). This builds on an intuition essential to the current language modeling paradigm, namely, that "language models are unsupervised multitask learners" (Radford et al., 2019). Thus, as in STaR, we leverage the LM's pre-existing reasoning ability to generate rationales and train the LM on them with a REINFORCE-based reward (Williams, 1992). We refer to this technique as Quiet-STaR, as it can be understood as applying STaR "quietly", training the model to think before it speaks.

Broadly, Quiet-STaR proceeds by generating rationales after every token to explain future text (*think*), mixing the future-text predictions with and without rationales (*talk*), and then learning to generate better rationales using REINFORCE (*learn*). We apply Quiet-STaR to Mistral 7B (Jiang et al., 2023) using the web text datasets OpenWebMath (Paster et al., 2023) and Colossal Clean Crawled Corpus (C4, Raffel et al. 2020). We find that, even without dataset-specific fine-tuning, Quiet-STaR results in improvements to zero-shot direct-reasoning abilities on CommonsenseQA (36.3%→47.2%) and GSM8K (5.9%→10.9%), and that these improvements consistently increase with the number of tokens used in the LM's internal thoughts. Lastly, we qualitatively investigate patterns in the generated rationales.

In solving this task, we make the following contributions:

1. We generalize STaR to learn reasoning from diverse unstructured text data. To our knowledge, this is the first work explicitly training LMs to **reason generally** from text, rather than on curated reasoning tasks or collections of reasoning tasks.
2. We propose and implement a **parallel sampling algorithm** that makes our training procedure scalable, generating rationales from all token positions in a given string.
3. We introduce custom **meta-tokens** at the start and end of each thought to allow the LM to learn that it should be generating a rationale and when it should make a prediction based on that rationale.
4. We apply a **mixing head** to retrospectively determine how much to incorporate the next-token prediction from a given thought into the current next-token prediction.
5. We show that a **non-myopic loss**, including multiple tokens ahead for language modeling, improves the effect of thinking.
6. On multiple tasks, we demonstrate that thinking allows the LM to predict difficult tokens better than one trained on the same web text, improving with longer thoughts.



(a) GSM8K (b) CommonsenseQA

Figure 2: **Generalization Results**. We evaluate the extent to which the model trained with Quiet-STaR generalizes to directly answering problems that require reasoning. The left plot (a) shows the zero-shot accuracy on GSM8K, while the right plot (b) shows the zero-shot accuracy on CommonsenseQA, without any fine-tuning. In both plots, the x-axis represents training steps, and each line corresponds to a different number of thinking tokens used during Quiet-STaR training. The y-axis measures the zero-shot direct accuracy on the respective datasets. We also include an inference normalized version of this plot in Figure 6.

2 Related Work

2.1 Reasoning in Language Models

There have been many works on training and exploiting language models to solve difficult tasks by first training them to reason through them. For example, Rajani et al. (2019) demonstrated that a pre-trained language model fine-tuned to output on human reasoning traces before answering multiple-choice commonsense reasoning questions outperformed one trained directly on answers. Shwartz et al. (2020) demonstrated that language models, when provided with some scaffolding, can generate these helpful chain-of-thought solutions without additional supervision. Later, Nye et al. (2021) demonstrated that "scratchpads" required less scaffolding when the language models were more capable, a result later reinforced by Wei et al. (2022b), emphasizing informal tasks, and further strengthened by Kojima et al. (2022), demonstrating this behavior could be accomplished zero-shot. Most recently, Wang & Zhou (2024) showed further that for commonsense-question answering, one could force a language model to leverage chain-of-thought reasoning by preventing it from emitting any valid answer tokens unless it was confident. However, once again, these approaches only work for a question-answer dataset, and Wang & Zhou (2024) relies on heuristics to identify when the model has output answer tokens. Somewhat like TRICE (Phan et al., 2023), we use the relative improvements in the log-likelihood of the target text across rationales as an estimate of quality, but we simply subtract the mean reward and do not incorporate more complex control variates.

2.2 Training Language Models to Reason

One direction that researchers have used to train language models to reason or improve their reasoning is training the language model on mined reasoning traces or reasoning-like data (Rajani et al., 2019; Wei et al., 2021a; Lewkowycz et al., 2022; Chung et al., 2022; Gunasekar et al., 2023). Although this approach has been demonstrated to be effective, it comes with drawbacks. It requires either manual annotation, which is sensitive to the capability of the annotators and is off-policy for the language model (i.e., the distribution of reasoning is not text that the language model would otherwise likely have generated). This approach is also expensive, difficult to scale, and provides no clear path to solving problems harder than those that the annotators are capable of solving.

Another direction for teaching reasoning relies on a language model's own generated reasoning, which can be seen as building on a large body of literature on self-play (Silver et al., 2017; Anthony et al., 2017; Polu & Sutskever, 2020). These include methods such as the

Algorithm 1: Quiet Self-Taught Reasoner (Quiet-STaR)

Input: Language model θ_0, training steps num_steps, sequence length l, thought length t, learning rate α, batch size b, number of thoughts $n_{thoughts}$, number of ground truth tokens used for supervising each thought n_{true}
Output: Language model θ that generates rationales to predict future text
for $i = 0$ **to** num_steps **do**
  Sample batch of sequences X of length l
  $h^{init} \leftarrow \text{hidden\_states}_{\theta_i}(X)$
  for $j = 1$ **to** l ***in parallel using attention mask*** **do**
    $\log p^{\text{init}}_{j:j+n_{true}} \leftarrow \text{lm\_head}_{\theta_i}(h^{init}_{j:j+n_{true}})$ `// Predict next tokens`
    $T_j \leftarrow \text{generate\_tokens}_{\theta_i}([X_{:j}; \texttt{<start\_thought>}], t, n_{thoughts})$ `// Generate thought`
    $T_j \leftarrow [T_j; \texttt{<end\_thought>}]$
    $h^{\text{thought}}_{j:j+n_{true}} \leftarrow \text{hidden\_states}_{\theta_i}([X_{:j}; T_j; X_{j:j+n_{true}-1}])$
    $\log p^{\text{thought}}_{j:j+n_{true}} \leftarrow \text{lm\_head}_{\theta_i}(h^{\text{thought}}_{j:j+n_{true}})$ `// Predict next tokens w/ thought`
    $w_{j:j+n_{true}} \leftarrow \text{mixing\_head}_{\theta_i}(h^{\text{thought}}_{j:j+n_{true}}, h^{init}_{j:j+n_{true}})$
    $\log p^{\text{talk}}_{j} \leftarrow w_{j:j+n_{true}} \cdot \log p^{\text{init}}_{j:j+n_{true}} + (1 - w_{j:j+n_{true}}) \cdot \log p^{\text{thought}}_{j:j+n_{true}}$ `// Mix logits`
    $\mathcal{L}^{\text{NLL}}_j \leftarrow -\log p^{\text{talk}}_{j:j+n_{true}}(X_{j+1:j+n_{true}+1})$
    $r_j = \log p^{\text{talk}}_{j:j+n_{true}}(X_{j+1:j+n_{true}+1}) - \log \overline{p}^{\text{talk}}_{j:j+n_{true}}(X_{j+1:j+n_{true}+1})$
    $\nabla_\theta \mathcal{L}^{\text{REINFORCE}}_j \leftarrow -r_j \mathbb{1}[r_j > 0] \cdot \nabla_\theta \log p_{\theta_i}(T_j | [X_{:j}; \texttt{<start\_thought>}])$
    $\nabla_\theta \mathcal{L}_j \leftarrow \nabla_\theta \mathcal{L}^{\text{NLL}}_j + \nabla_\theta \mathcal{L}^{\text{REINFORCE}}_j$
  $\theta_{i+1} \leftarrow \theta_i - \alpha \sum_{j=1}^{l} \nabla_\theta \mathcal{L}_j$ `// Update model parameters`
return $\theta_{\text{num\_steps}}$

Self-Taught Reasoner (Zelikman et al., 2022), which demonstrated that a language model iteratively trained on its reasoning that led to correct answers could solve increasingly difficult problems. Later work aimed to leverage additional information or assumptions such as Huang et al. (2022) which demonstrated that the algorithm proposed in STaR could still work if one assumed that the majority-vote answer was correct (although this has a lower ultimate performance). Further work has generalized the results of Zelikman et al. (2022), such as Uesato et al. (2022) which demonstrated additional usefulness to "process-based" supervision where incorrect reasoning traces were filtered, recently V-STaR (Hosseini et al., 2024) that demonstrates that training a verifier to guide generation also improves performance, as well as TRICE (Hoffman et al., 2024) which maximizes the marginal likelihood of the correct answer given several reasoning traces per problem. Finally, related work has also explored learning intermediate reasoning in the constrained setting of making mathematical statements, where statements in the model's intermediate reasoning could be constrained to only be valid mathematical statements (Poesia et al., 2023). We include further discussion of related reasoning works in Appendix F.

2.3 Meta-tokens

Recently, a growing body of work has demonstrated the usefulness of custom tokens optimized to perform specific functions in the context of a neural network – for this reason, they have also been referred to as "function vectors." (Todd et al., 2023). One of the original instantiations of this was prompt-tuning (Lester et al., 2021) (and relatedly prefix-tuning (Li & Liang, 2021)), where the embeddings corresponding to the tokens of a prompt could be optimized to better accomplish a task. Others have applied meta-tokens to compress long prompts (Li et al., 2023; Jung & Kim, 2023) for efficiency. Most relevant to this work, Mu et al. (2024) optimized a token such that, when the tokens after it could not attend to the tokens before it (i.e., a context compression token), it would provide sufficient information to future tokens. Although we do not focus on compression, we share the problem of learning a

token that affects attention and controls complex downstream behavior. In one related work, Goyal et al. (2023) show that learning a single "pause" token (essentially representing each token as two tokens) improves LM performance. However, unlike the thought tokens in our work, this pause token does not initialize a thought – instead, it can be seen as acting as the entirety of the thought. We find that reasoning in language is significantly more helpful.

3 Problem Statement

In this work, we introduce an auxiliary 'rationale' variable between each pair of observed tokens of the sequence. We then aim to optimize a language model with parameters θ with the capacity to generate intermediate thoughts (or rationales) such that

$$\theta^* = \arg\max_\theta E_x \left[log\, p_\theta \left(x_{i:n} | x_{0:i}, \text{rationale}_\theta \left(x_{0:i} \right) \right) \right]$$

Note that, in principle, this provides no advantage over an optimal language model that already correctly models the language's distribution over strings. Yet, in practice, extensive prior work has shown that language models benefit from intermediate rationales on reasoning tasks (Nye et al., 2021; Zelikman et al., 2022; Wei et al., 2022b). Some work has aimed to explain the effects of chain-of-thought reasoning, namely attributing it to "locality of experience" (Prystawski et al., 2024). More broadly, reasoning allows a model to decompose a challenging computation into smaller steps. In effect, we train the model to learn which decomposition and planning steps are effective in predicting future text. Also note that we formulate the objective as accurately predicting the remaining sequence, rather than only the next token. Once again, for an optimal LM these would be equivalent. However we find that the non-myopic formulation leads to a more effective loss for learning rationales.

4 Quiet-STaR

4.1 Overview

Quiet-STaR operates with three main steps (Figure 1):

1. **Parallel rationale generation (think, Subsection 4.2)**: In parallel across n tokens x_i in an input sequence $x_{0:n}$, we generate r rationales of length t: $c_i = (c_{i1}, \ldots, c_{it})$, resulting in $n \times r$ rationale candidates. We insert learned <|startofthought|> and <|endofthought|> tokens to mark each rationale's start and end.
2. **Mixing post-rationale and base predictions (talk, Subsection 4.3)**: From the hidden state output after each rationale, we train a "mixing head" – a shallow MLP producing a weight determining how much the post-rationale next-token predicted logits should be incorporated compared to the base language model predicted logits. This approach eases distribution shift early in finetuning, due to introducing rationales.
3. **Optimizing rationale generation (learn, Subsection 4.4)**: We optimize the rationale generation parameters (start/end tokens and LM weights) to increase the likelihood of rationales that make future text more probable. We use REINFORCE to provide a learning signal to rationales based on their impact on future-token prediction. To reduce variance, we apply a teacher-forcing trick to include in the loss the likelihood of predicting not only the token after the thought but also later tokens.

4.2 Parallel Generation

A key challenge in Quiet-STaR is efficiently generating rationales at each token position in the input sequence. Naively, this would require a separate forward pass for each token, which becomes computationally intractable for long sequences.

We allow for highly parallel generation by first observing that an inference pass of a language model produces a probability distribution over the next tokens for all input tokens. Naturally, this allows us to sample one next token from each token in the input. If one has generated a successor from each token, it is not possible to simply continue with the original sequence. For example, imagine predicting the next token after each token of "$<bos>$ the cat sat" one might generate "yes orange saw down" – each successor by itself is a reasonable next token

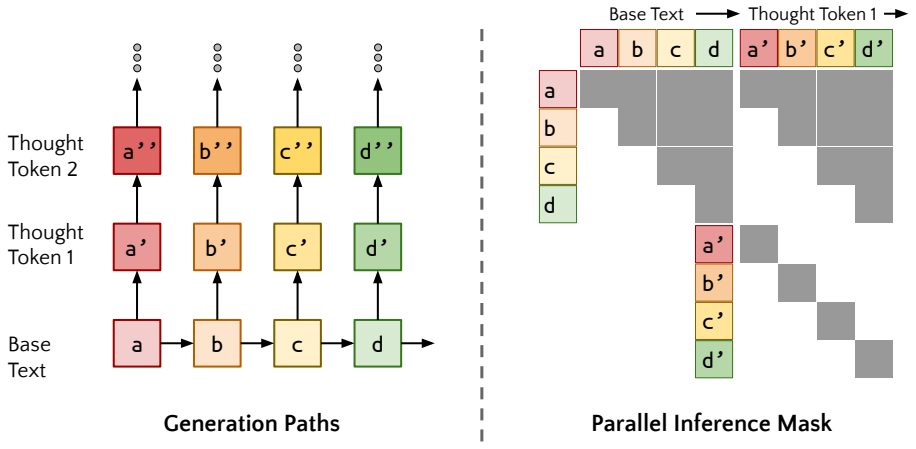


Figure 3: **Parallel Generation**. By constructing an attention mask that allows all thought tokens to pay attention to themselves, all preceding thought tokens within the same thought, and the preceding text, we can generate continuations of all of the thoughts in parallel. Each inference call is used to generate one additional thought token for all text tokens.

to a prefix of the sequence, but the list of tokens is a set of "counterfactual" continuations of these prefixes. We can, however, leverage these continuations to generate hidden thoughts for each observed token.

To do this efficiently, we cache each forward pass and concatenate a diagonal attention mask to the previous attention mask: each generated token now attends to all of the tokens that were used to generate it, as well as to itself (but not to token on other "counterfactual" paths). Moreover, this parallelized next-sampling token procedure can be repeated arbitrarily many times (or at least, until one runs out of memory). We visualize this procedure in Figure 3 and highlight additional ways to make this algorithm faster in Appendix B.

4.3 "Mixing" (Residual) Heads

When starting with a pre-trained model, thoughts will initially be out of distribution, and hence harm language modeling performance. To smooth the transition to thinking, we introduce a learned interpolation between the LM predictions with and without thoughts. Given the end-of-thought token's hidden state and the hidden state of the original text token, the mixing head outputs a weight that determines the extent to which the post-thought prediction logits will be used. We use a shallow multi-layer perceptron for this head, outputting a scalar for each token. We include implementation details in Appendix A.

4.4 Optimizing Rationale Generation

4.4.1 Optimizing Start-of-Thought and End-of-Thought Tokens

The <|startofthought|> and <|endofthought|> tokens serve as learned meta-tokens that control the model's rationale generation. Optimizing the representation of these tokens, especially the <|startofthought|> token, is crucial but challenging due to the discrete nature of the rationale tokens. We initialize the start and end token embeddings to the embedding corresponding to the em dash, "---", which often appears in text data to denote a pause or thought. This leverages the language model's preexisting knowledge. In addition, to allow these embeddings to be optimized more quickly, we apply a (hyperparameter) weight to the gradients of these embeddings during the update step. Intuitively, the start thought tokens can be understood as putting the model into a "thinking mode" and the end thought token can be understood as telling the model when it's done thinking.

4.4.2 Non-myopic Scoring and Teacher-forcing

Because we do not expect thoughts to be useful in predicting every token, we would prefer the model's reward to depend less on the exact next word in the text following the thought and more on the following semantic content. There are two primary challenges here. First, unlike in typical language modeling with transformers, only the thoughts corresponding to

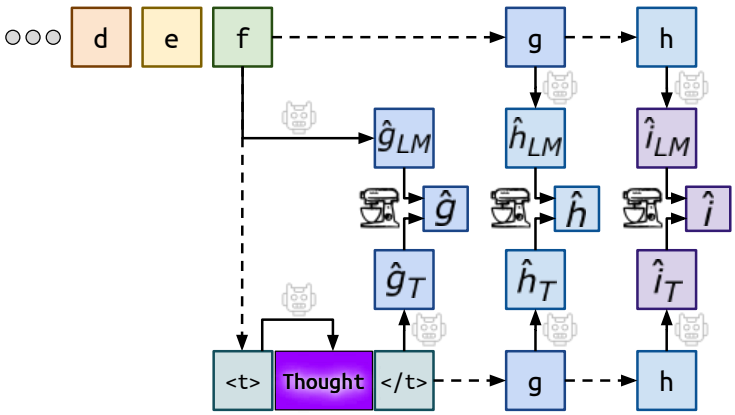


Figure 4: **Forward Pass and Teacher Forcing**. We visualize a single forward pass of our algorithm. Solid lines denote language model computation, while dashed lines indicate tokens are inserted via teacher forcing, and the mixer represents the mixing head. In particular, we visualize predicting three tokens ahead. Thought generation is shown in more detail in Figure 1 and Figure 3.

a given next-token prediction receive a gradient from that prediction—a consequence of our parallel sampling strategy. We could address this by adding loss terms for future tokens by sampling the tokens before. However this would result in much higher entropy for language modeling in general and lower-quality generated text, because it would train the LM to partially disregard its preceding tokens. Instead, we use the parallel attention mask to compute the log probabilities of the true next tokens, applying teacher forcing by assuming the model selected the correct next ground-truth token (as implicit in normal language modeling with transformers). Note that the loss for each future token also depends on a mixing weight computed from the end thought token and the previous observed token. The number of future tokens included in the loss is a hyper-parameter. We apply the same teacher-forcing technique to insert the start and end tokens. We visualize this procedure in Figure 4.

4.4.3 Objective

We use REINFORCE to optimize the likelihoods of the rationales based on their usefullness: the log-likelihood of the n_{true} true next tokens $X_{j+1:j+n_{true}+1}$ under the language model given previous observed tokens and a particular rationale ($p^{\text{talk}}_{j:j+n_{true}}$ as shorthand for the mixed prediction probabilities after thinking, see Algorithm 1). To reduce variance, we generate multiple rationale continuations for each token in the input sequence (loosely inspired by TRICE, Phan et al. (2023)). We thus define the reward r_j for each rationale T_j as the difference between $p^{\text{talk}}_{j:j+n_{true}}$ and the average across rationales for that token ($\overline{p}^{\text{talk}}_{j:j+n_{true}}$):

$$r_j = \log p^{\text{talk}}_{j:j+n_{true}}(X_{j+1:j+n_{true}+1}) - \log \overline{p}^{\text{talk}}_{j:j+n_{true}}(X_{j+1:j+n_{true}+1})$$

We then use this reward in a REINFORCE loss term to update the language model parameters θ to increase the likelihood of rationales that perform better than the average:

$$\nabla_\theta \mathcal{L}^{\text{REINFORCE}}_j = -r_j \cdot \nabla_\theta \log p_\theta(T_j | [X_{:j}; \texttt{<|startofthought|>}])$$

We found it useful to exclude the negative reward from the REINFORCE loss term, as it led to more stable training, though it may introduce some bias.

This loss term encourages the model to generate rationales that improve its predictions of future tokens compared to the average prediction across all generated rationales for that token. The gradients from this loss are used to update both the LM parameters and the start-of-thought and end-of-thought token embeddings, with a (hyperparameter) weight applied to the gradients of the start-of-thought and end-of-thought token embeddings to accelerate their optimization. By iteratively optimizing these parameters, Quiet-STaR trains the model to generate more useful rationales throughout training. Lastly, we also include a log-likelihood loss, $\mathcal{L}^{\text{NLL}}_j$, to ensure that the LM learns to optimize the talking heads and also receives a next-token prediction signal for the base LM head[1].

[1]Due to our linear mixing, equivalent to shifting the mixing weight toward the base prediction.

5 Experiments and Results

Intuitively, not all tokens require equal amounts of thought. For example, consider the sentence "the person is run-": although there is inevitably some probability of the token being something other than "ing"[2], as a standalone sentence without context, additional thinking is unlikely to improve a well-trained model's prediction. Indeed, we conjecture that for most chunks of most online text, additional thought has little to no impact. Indeed, early in our exploration we observed that Quiet-STaR does not benefit all tokens equally. Thus, we design our experiments to investigate whether our approach is useful in predicting tokens that *do* require thought. We evaluate 1) whether Quiet-STaR improves a language model's ability to directly predict answers in datasets that require reasoning; and, 2) the distribution of impacts resulting from thinking tokens. We conduct all of our experiments starting with the base version of Mistral 7B (Jiang et al., 2023).

We perform most of our experiments by training on OpenWebMath (Paster et al., 2023), a crawl that emphasizes more technical webpages. We selected OpenWebMath because we anticipated that it would have a higher density of tokens that benefit from reasoning, which our experiments support. We also evaluate Quiet-STaR on C4 (Raffel et al., 2020), a widely used LM pretraining corpus with more diverse text, and again show significant albeit smaller benefits.

5.1 Downstream Performance

In this subsection, we evaluate the extent to which Quiet-STaR improves the zero-shot reasoning capabilities of the language model on CommonsenseQA (Talmor et al., 2018) and GSM8K (Cobbe et al., 2021). On CommonsenseQA, we find that Quiet-STaR improves performance by 10.9% compared to the base language model. As shown in Figure 2, this improvement consistently increases with the number of tokens used in the model's rationales, indicating that more thorough reasoning through the thought tokens is translating to better direct question-answering performance. Similarly, on GSM8K, Quiet-STaR results in a 5.0% boost over the base model, and once again, performance scales with the length of the rationales generated during Quiet-STaR training. For reference, in Figure 2, we include a baseline corresponding to training the same model on the same dataset without thought tokens. We observe that in multiple curves, performance appears to eventually deteriorate – we anticipate that this is because we are not training on these downstream tasks, so the roles of the thought tokens may change over time. We also find a benefit of our non-myopic objective, which we discuss in Appendix D.

We find that training with Quiet-STaR on C4 (Raffel et al., 2020) also improves performance on GSM8K (5.9% $\rightarrow$ 8.1%) and CommonsenseQA (36.3% $\rightarrow$ 42.6%) but by a smaller margin. Specifically, for our C4 evaluation, we train Mistral 7B with 16 thought tokens and 4 true tokens ahead and otherwise the same setup.

We can compare these improvements to those offered by pause tokens (Goyal et al., 2023), which can be seen as a constrained version of Quiet-STaR where each token is represented by two tokens and the second "pause" token acts as the entirety of the thought. In particular, our setup is most comparable to their pause token fine-tuning, as we also finetune a pretrained model. Their results indicate that pause token fine-tuning also provides minor gains over the base model on CommonsenseQA, they observed an improvement from 26.9% to 28.8%; on GSM8K, Goyal et al. (2023) found that pause token fine-tuning harms performance. Moreover, on both tasks (and the majority of their evaluated tasks), they observed that additional thought tokens harmed performance. Moreover, they discuss the "lukewarm effect of pause-finetuning a standard-pretrained model" (Goyal et al., 2023). This suggests that allowing the model to generate multi-token rationales leads to more effective reasoning compared to the single-token "pauses". Note however, that unlike Goyal et al. (2023), we *do not fine-tune* on the downstream tasks.

Overall, these downstream results validate that training a language model to predict the subtext between the lines of general text data can substantially improve its reasoning

[2]For example, in this very text, the token following "run" is "-"

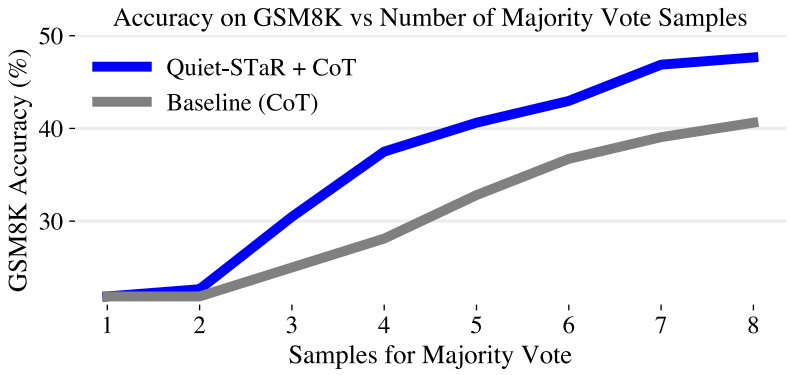


Figure 5: **Zero-shot performance on Quiet-STaR applied to chain-of-thought on GSM8K**. We visualize how using a Quiet-STaR trained Mistral model can improve chain-of-thought performance. We use an 8-thought-token-trained model and use its internal thoughts to improve the tokens in a zero-shot chain-of-thought (Kojima et al., 2022)

capabilities, even on datasets it was not explicitly trained on. The fact that longer rationales consistently lead to better outcomes, and that Quiet-STaR outperforms the constrained pause token approach, supports the notion that Quiet-STaR is successfully teaching the model to leverage its own generated thoughts to reason more thoroughly about the input.

5.2 Improvement Distribution

As visualized in Appendix Figure 7, we find that on average there is little improvement in the LM's ability to predict arbitrary tokens. But, when we visualize the distribution of relative improvements, there is a disproportionate improvement on more difficult tokens. This reflects the idea that some text tokens are substantially harder and benefit more from careful thought.

In Appendix Figure 8, we aim to provide some insight into the kinds of tokens where the improvements occur. Namely, while thinking appears to help for many tokens in the example, inspection suggests it disproportionately help to predict tokens where recalling relevant information is useful, such as the name of an applicable theorem or the start of the next step in a proof. Notably, this would align well with the framing proposed by Prystawski et al. (2024).

5.3 Quiet-STaR and Chain-of-Thought

While there are natural parallels between chain-of-thought prompting and our approach, they are orthogonal and complementary. In zero-shot chain-of-thought, a user actively prompts the model to think 'out loud', otherwise using its ordinary production distribution (Kojima et al., 2022); Quiet-STaR instead allows a model to think quietly at every token, with a distribution trained to be useful. We investigate using silent, Quiet-STaR, rationales while generating explicit CoT reasoning. Because our goal is generalist reasoning that requires no task-specific input at all, we used a zero-shot prompt ("Let's think step by step.") without in-context examples. Our experiments indicate that internal rationales allow the model to generate more structured and coherent chains of thought, shown in Appendix E and visualized in Figure 5. The majority vote accuracy over 8 samples (cot-maj@8) increases from 40.6% to 47.7% with Quiet-STaR, as evaluated on a sample of 128 GSM8K test items. Note that each chain-of-thought solution is sampled with temperature 0.7.

5.4 Examples

While there is no explicit regularization in Quiet-STaR for thoughts to be human-interpretable, they are generated from the same transformer trained to model language, hence likely to be at least partially understandable. We discuss why this design choice benefits the training stability in Appendix I. For reference, we include examples of thoughts generated that were helpful to the model in predicting future tokens in OpenWebMath. First, in one case, recalling that one should start with magnesium to produce magnesium nitride allows it to better predict that the first step of the procedure involves heating magnesium.

```
'<s> # Magnesium reacts with nitrogen to form magnesium nitride. The chemical
    formula for this reaction is Mg+N_2-> MgN_2. What is the product, or what
    are the products, of this reaction?\n\nJan 12, 2016\n\nThe formula for
    magnesium nitride is $M {g}_{3} {N}_{2}$.\n\n#### Explanation:\n\nAs do
    many active metals, magnesium nitride can be<|startofthought|> 1 --, so the
     equation of the reaction that forms magnesium nitride is\n\n$Mg + N_2 \\to
    <|endofthought|> formed by heating the metal (fier'
```

In some cases, the most useful thoughts appear to be near-continuations that correspond more closely to the target text, e.g.,

```
An integer $n$ is odd if $n = 2k+1$ for some integer $k$.\n\nTo prove that $A =
     B$, we must show that $A \\subseteq B$ and $B \\subseteq A$. The first of
    these tends to<|startthought|> in some sense - to be the more difficult<|
    endthought|> trickiest for students
```

Lastly, we include an example from answering CommonsenseQA. Notably, this thought occurs while reading the question and hence was not used to predict the final answer.

```
'<s> Q: Talking to the same person about the same thing over and over again is
    <|startofthought|>\n\n(a) a one-to-one correlation\n\n(b) a one-to<|
    endofthought|> something someone can what?'
```

6 Limitations

This work proposes a new framework for learning to reason, and in doing so explores solutions to a variety of meta-learning challenges. However, to solve these challenges, certain simplifications were necessary. For example, it would be valuable to understand whether these techniques work when a model is trained from scratch. We have also only applied Quiet-STaR to a 7 billion parameter model, albeit a powerful one. The same techniques applied to a better model would likely yield disproportionately better results, as has often been observed for gains from reasoning (Wei et al., 2022a).

Quiet-STaR results in a substantial overhead, generating many tokens before generating every additional token. (See Appendix C for compute adjusted performance results.) However, this can also be seen as an advantage: typically, a language model can generate the next token based on the current context, and while there are techniques to improve sampling quality, there is no general way to leverage additional compute to enhance next-token prediction. In the current implementation we do not support dynamically predicting when to generate, or end, a rationale. However, this would be a natural extension. For instance, if the mixing head was a prediction from the base language model, before any thought, rather than after the thought, one could apply a threshold to prevent generating thoughts that would not be incorporated. We expect that this is a more difficult task, as predicting the usefulness of a thought is simpler when one has already generated the thought.

7 Conclusion

Quiet-STaR represents a step towards language models that can learn to reason in a general and scalable way. By training on the rich spectrum of reasoning tasks implicit in diverse web text, rather than narrowly specializing for particular datasets, Quiet-STaR points the way to more robust and adaptable language models. Our results demonstrate the promise of this approach, with Quiet-STaR improving downstream reasoning performance while generating qualitatively meaningful rationales. We believe this also opens many potential future directions - for example, one may aim to ensemble thoughts in order to further improve the predictions for future tokens. Moreover, if the language model can predict when thought will be useful, for example by putting the mixing head before the prediction, then the predicted mixing weight could be used to dynamically allocate compute during generation. Future work can build on these insights to further close the gap between language model and human-like reasoning capabilities.