arXiv:2407.10930v2 [cs.CL] 7 Oct 2024

Fine-Tuning and Prompt Optimization: Two Great Steps that Work Better Together

Dilara Soylu **Christopher Potts** **Omar Khattab**

Stanford University

`{soylu,cgpotts,okhattab}@stanford.edu`

Abstract

Natural Language Processing (NLP) systems are increasingly taking the form of sophisticated modular pipelines, e.g., Retrieval Augmented Generation (RAG), where each module may involve a distinct Language Model (LM) and an associated prompt template. These compound systems often lack intermediate labels or gradient flow to optimize each module, making their end-to-end optimization challenging. Here we seek strategies to optimize both the module-level LM weights and the associated prompt templates of such systems to maximize a downstream task metric. We propose for the first time combining the weight and prompt optimization strategies to optimize a modular LM pipeline by alternating between the two to get the same LM to teach itself. In experiments with multi-hop QA, mathematical reasoning, and feature-based classification using `mistral-7b`, `llama-2-7b`, and `llama-3-8b`, these **`BetterTogether`** strategies optimizing the weights and prompts of a pipeline together outperform directly optimizing weights alone and prompts alone by up to 60% and 6%, respectively, on average across LMs and tasks. Our **`BetterTogether`** optimizer is released in DSPy at `http://dspy.ai`.

1 Introduction

While the capabilities of language models (LMs) continue to grow, recent work has shown the potential of building more powerful Natural Language Processing (NLP) systems by *composing* multiple skills of LMs into pipelines. Examples of this include systems for retrieval-augmented generation (Guu et al., 2020; Lewis et al., 2020; Ma et al., 2023; Jiang et al., 2023b), multi-hop reasoning (Qi et al., 2021; Khattab et al., 2021), information extraction (Pourreza and Rafiei, 2023; D'Oosterlinck et al., 2024), and other sophisticated pipelines (Zelikman et al., 2022; Dohan et al., 2022; Khattab et al., 2022; Beurer-Kellner et al., 2023; Schlag et al., 2023; Viswanathan et al., 2023).

Such *LM Programs* offer much more control for designing NLP systems, as they break down problems into modular, more manageable sub-tasks that can be assigned to LMs. If we could teach these LMs to accurately conduct their easier subtasks and to communicate effectively within multistage pipelines, this could greatly expand the scope of reliable NLP systems we can build.

To this end, Khattab et al. (2024) introduced the DSPy framework for defining and *optimizing* LM Programs. In it, a program is defined as a function Φ that composes a set of stages, which we will refer to as *language modules* $M = \langle M_1, \ldots, M_{|M|} \rangle$, into a pipeline. Each language module M_i specifies a fuzzy natural-language transformation (e.g., generating a summary of a supplied document) that needs to be learned. To do so, each module learns a particular prompt (template) π to make a call to a particular LM with weights θ. The optimization problem is then defined as maximizing the expected performance (per a downstream metric μ) of the program Φ over a set of inputs by updating each module's π and θ.

Existing work (Khattab et al., 2024; Opsahl-Ong et al., 2024) has studied optimizing the discrete string prompt of each module and has considered simple approaches for fine-tuning each module's LM weights. In this empirical study, we investigate *updating each module's LM weights and prompt template together* to maximize a downstream metric on the final output of the program. Doing this is challenging as Φ is not generally differentiable and its modules M_i generally lack labeled outputs, while exhibiting sophisticated dependencies. Moreover, in realistic settings, the training set is usually very small and only a small number of LM calls are possible for training and inference.

To address this challenge, we propose to *alternate between fine-tuning LM weights and optimizing prompt templates* and evaluate approximate optimization strategies in which we bootstrap training labels for all pipeline modules. In experiments with multi-hop QA (`HotPotQA`), mathematical reasoning (`GSM8K`), and feature-based classification (`Iris`), we show that these tandem strategies are highly effective across three different LMs, leading to 5–78% gains for `HotPotQA`, 2.5–10% gains for `GSM8K`, and 3.5–88% gains for `Iris` against prompts only and weights only strategies, averaged across `mistral-7b-instruct-v0.2`, `llama-2-7b-chat`, and `llama-3-8b-instruct`.

2 Problem Statement

We are given an LM program Φ, which operates like a blackbox function $\Phi : \mathcal{X} \to \mathcal{Y}$, in which $\mathcal{X}$ and $\mathcal{Y}$ are typically in natural language (e.g., questions and their program-generated answers, respectively). For example, we may have a program Φ for answering complex questions with short factoid answers. In the course of its execution, Φ makes one or more calls to each of its $|M| \geq 1$ *language modules*, $M = \langle M_1, \ldots, M_{|M|} \rangle$.

For example, the program may implement a *multi-hop, retrieval-augmented* pipeline for question answering. This common pipeline (Qi et al., 2021; Khattab et al., 2021; Press et al., 2023; Khattab et al., 2022) breaks down the input into subquestions that are used to iteratively find relevant passages (e.g., from a corpus like Wikipedia) until the question can be faithfully answered. In general terms, each module $M_i : \mathcal{X}_i \to \mathcal{Y}_i$ is a *declarative* LM invocation that defines, in inherently fuzzy natural-language terms, an input $\mathcal{X}_i$ domain (like a *user-supplied question* and *a set of retrieved passages*) and an output $\mathcal{Y}_i$ co-domain (like a *search query to find additional relevant passages*).

We seek to implement each language module as some specific, well-tuned strategy for invoking an underlying language model **LM**. Concretely, we assume that a module M_i will be fully implemented by specifying (1) the string prompt π_i in which the module inputs $\mathcal{X}_i$ are plugged in to decode the module outputs $\mathcal{Y}_i$ and (2) the floating-point weights θ_i assigned to the parameters of **LM** in the course of this module. We refer to the version of Φ in which the prompts and LM weights are assigned explicitly to Π and Θ, respectively, as $\Phi_{\langle \Theta, \Pi \rangle}$.

Given nothing but a small training set $X = \{(x_1, m_1), \ldots, (x_{|X|}, m_{|X|})\}$ of inputs $x_i \in \mathcal{X}$ and optional *metadata* like output labels or other *hints* $m_i \in \mathcal{M}$ that can be used for determining the correctness of a given program run, and a metric $\mu : \mathcal{Y} \times \mathcal{M} \to \mathbb{R}$, our goal is to optimize Φ, that is, configure its modules' prompts and LM weights to maximize the following objective.

$$\arg\max_{\Theta,\Pi} \frac{1}{|X|} \sum_{(x,m)\in X} \mu(\Phi_{\langle \Theta, \Pi \rangle}(x), m)$$

Researchers tuning LM pipelines are in effect seeking to achieve this objective. It is also a very large subspace of the optimization problem in the DSPy framework[1] for LM programs. Unfortunately, this problem is intractable: the search space is large and we don't have gradients or intermediate output labels to optimize each module, so we seek approximate strategies for such optimization.

3 `BetterTogether`: Alternating Weight and Prompt Optimization Steps for LM Programs

We now introduce the `BetterTogether` algorithm, which alternates the weight and prompt optimization steps for LM programs. We hypothesize that, when a large LM is used to teach itself how to tackle the task defined by an LM program, fine-tuning LM weights and prompts are *both* essential to achieve the highest quality. In particular, we expect that (1) prompt optimization before fine-tuning can lead to more successful datapoints for fine-tuning, and, (2) prompt optimization after fine-tuning can make adjustments to the behavior of the LM program, leading to higher quality outputs. Considering that fine-tuning is often perceived as a more powerful tool, this can be surprising, especially when both approaches are ultimately applied over the same set of training inputs X.

Algorithm 1 `BetterTogether`: Optimizing LM programs by alternating prompt and weight optimization steps, instantiated in Algorithm 2

```
Input: Program $\Phi_{\langle \Theta, \Pi \rangle} = \Phi_\Theta \odot \Phi_\Pi$,
         with module weights $\Theta = [\theta_1, \ldots, \theta_{|\Phi|}]$
         and module prompts $\Pi = [\pi_1, \ldots, \pi_{|\Phi|}]$
  Training Set $X$ and Metric $\mu$
function BetterTogether($\Phi_{\langle \Theta, \Pi \rangle}$, $X$, $\mu$)
    $\Pi' \leftarrow$ OptimizePrompts($\Phi_{\langle \Theta, \Pi \rangle}$, $X$, $\mu$)
    $\Theta' \leftarrow$ FinetuneWeights($\Phi_{\langle \Theta, \Pi' \rangle}$, $X$, $\mu$)
    $\Pi'' \leftarrow$ OptimizePrompts($\Phi_{\langle \Theta', \Pi \rangle}$, $X$, $\mu$)
    return $\Phi_{\langle \Theta', \Pi'' \rangle}$
end function
```

Accordingly, the general optimization framework for our algorithm is defined in Algorithm 1.

---

[1] `http://dspy.ai`

Given a program Φ, the algorithm begins by optimizing Φ's prompts, then fine-tuning its set of LM weights with the data bootstrapped using the optimized prompts, and finally optimizing its prompts again using the fine-tuned weights. In principle, each of these steps could be treated as optional. This will define the different possible combinations of **`BetterTogether`** that we will seek to evaluate in Section 4. Specifically, we are interested in the quality of (1) the vanilla program Φ with simple user-supplied instructions as the prompts and no fine-tuning of **LM**, (2) optimizing the prompts only, (3) optimizing the weights only, (4) optimizing the prompts twice, i.e., using the prompt-optimized Φ as a starting point for a second round of prompt optimization, (5) optimizing the weights twice, (6) optimizing the prompts then the weights, (7) vice versa, and (8) optimizing the prompts, weights, then prompts. Overall, we expect the final three to consistently outperform the first five.

For Algorithm 1 to be complete, we need to instantiate Lines 1–3 with specific approaches for prompt optimization and LM fine-tuning. For this, we choose the `Bootstrap-*` family of algorithms from Khattab et al. (2024), which work by executing an initial version of the program on input examples $(x_i, m_i) \in X$ and recording the inputs/outputs observed at each module when the *final output* is "correct", i.e., $\mu(\Phi(x_i), m_i) \geq \lambda$ for some threshold λ (e.g., 1.0 for binary accuracy). This is important to note: in line with our problem formulation, our prompt and weight optimization regimes are not simply training on hand-labeled data but on self-generated program traces.

Instantiations for Lines 1–3 of Algorithm 1 are shown in Algorithm 2. For prompt optimization, we use `BootstrapFewshotRS` (`BFRS`) of DSPy, which self-generates potential few-shot examples of every module and applies a form of random search (`RS`) to select the specific generated few-shot examples that are used for prompting. Overall, `BFRS` first divides X into a training split T and a validation split V (Line 2). It then executes the provided $\Phi_{\langle\Theta,\Pi\rangle}$ on the training inputs, collecting input–output pairs for every module in Φ for each $x_i \in T$. This is called a trace τ, and we keep only the traces assigned high scores by μ (Line 4). Given all of these traces, `BFRS` samples multiple different subsets of a few traces τ' (Line 6), each of them containing a potential few-shot example for *each* module in Φ, and ultimately selects the subset that, when used to construct few-shot prompts (Line 7), achieves the highest score (Line 8). This simple search strategy is known to consistently lead to large quality improvements in prompting LM programs (Khattab et al., 2024; Opsahl-Ong et al., 2024), often outperforming manually or automatically optimizing prompt instructions or writing examples by hand.

Algorithm 2 Instantiating Algorithm 1's prompt & weight optimizers with bootstrapping algorithms

```
Input: Training Set $X$ and Metric $\mu$
 1: function BootstrapFewShotRS($\Phi_{\langle\Theta,\Pi\rangle}$, $X$, $\mu$)
 2:     $T, V \leftarrow$ SplitIntoTrainAndValidation($X$)
 3:     $\tau \leftarrow$ BootstrapTraces($\Phi_{\langle\Theta,\Pi\rangle}$, $T$)
 4:     $\tau \leftarrow$ FilterTraces($\tau$, $\mu$)
 5:     Initialize attempts list $\mathcal{A} \leftarrow \{\}$
 6:     for $\tau' \in$ SampleFewShotSubsets($\tau$) do
 7:         $\Pi' \leftarrow$ ConstructFewShotPrompts($\tau'$)
 8:         $\sigma \leftarrow \frac{1}{|V|} \sum_{\langle x_i, m_i \rangle \in V} \mu(\Phi_{\langle\Theta,\Pi'\rangle}(x_i), m_i)$
 9:         Extend $\mathcal{A}$ with $(\sigma, \Pi')$
10:     end for
11:     return $\Pi_{\max}$, $\mathcal{A}$'s highest-scoring prompts sequence
12: end function
13:
14: function BootstrapFinetune($\Phi_{\langle\Theta,\Pi\rangle}$, $X$, $\mu$)
15:     $\tau \leftarrow$ BootstrapTraces($\Phi_{\langle\Theta,\Pi\rangle}$, $X$)
16:     $\tau \leftarrow$ FilterTraces($\tau$, $\mu$)
17:     $\Theta' \leftarrow$ TrainLM($\tau$)
18:     return $\Theta'$
19: end function
20:
21: Set OptimizePrompts as BootstrapFewShotRS
22: Set FinetuneWeights as BootstrapFinetune
```

For fine-tuning, we extend `BootstrapFinetune` (`BFT`) of DSPy, which, given a program Φ, self-generates a large number examples for every module and combines them into one dataset to fine-tune the LM weights with an implicit multi-task objective, where the sub-tasks are the modules' roles. Existing work has only considered `BFT` in a very narrow setting for LM programs: on `HotPotQA`, Khattab et al. (2024) train a `T5-Large` model using traces from a few-shot `Llama2-13b` program, without considering getting an LM to teach itself via `BFT` nor considering a role for `BFRS` in the fine-tuned program. In this work, we focus on allowing models to teach themselves and self-improve. We propose for the first time combining the strategies of `BFRS` and `BFT` via alternation to get the same LM to teach itself better than either prompt or weight optimization in isolation. One could test similar ideas in scenarios where a larger model does the bootstrapping for a smaller LM. This may lead to even better results but is outside our scope.

Strategy	`mistral-7b-instruct-v0.2`			`llama-2-7b-chat`			`llama-3-8b-instruct`		
	`HotPotQA`	`GSM8K`	`Iris`	`HotPotQA`	`GSM8K`	`Iris`	`HotPotQA`	`GSM8K`	`Iris`
Baseline Strategies									
Vanilla Zero-shot	17.2	40.3	26.0	13.2	24.0	0.0	31.6	72.7	48.0
Prompt Optimization (Π)	33.8	46.4	57.3	33.3	26.0	56.7	**46.9**	77.9	79.3
Weight Optimization (Θ)	22.9	40.7	29.3	12.2	24.0	–	34.8	75.1	37.3
$\Pi \rightarrow \Pi$	33.8	47.7	59.3	32.6	24.7	64.0	46.5	77.6	**82.0**
$\Theta \rightarrow \Theta$	24.0	42.8	38.0	13.0	24.1	–	34.4	44.1	39.3
BetterTogether Strategies									
$\Pi \rightarrow \Theta$	36.3	47.3	30.7	32.7	**27.3**	26.7	42.8	77.6	44.0
$\Theta \rightarrow \Pi$	33.0	**48.3**	**66.7**	34.2	26.6	–	43.6	**78.9**	78.7
$\Pi \rightarrow \Theta \rightarrow \Pi$	**37.6**	46.8	52.7	**34.8**	26.3	**65.3**	46.7	77.0	79.3

Table 1: Main Results. Percentage accuracies of baseline and **`BetterTogether`** strategies on `HotPotQA`, `GSM8K`, and `Iris` evaluated on `mistral-7b-instruct-v0.2`, `llama-2-7b-chat`, and `llama-3-8b-instruct`. Reported are average performance of 3 runs on a held-out test set using different random seeds. **Bold** font is used to mark the highest score in a given column. Strategies where weight optimization is the first step use the vanilla (zero-shot) strategy to generate the initial fine-tuning dataset. If a model generates very few or no correct outputs under the vanilla strategy on the training set used to bootstrap the fine-tuning data, there will not be a sufficient dataset for fine-tuning. These settings are marked with "–".

4 Experimental Evaluation

We now seek to evaluate our hypothesis on the importance of optimizing both LM weights and prompts of LM programs. We conduct our evaluation across three datasets that span different tasks (and thus LM programs) each. In particular, we use `HotPotQA` (Yang et al., 2018) for multi-hop reasoning, `GSM8K` (Cobbe et al., 2021) for arithmetic reasoning, and `Iris` (Fisher, 1988) for classification. We run our experiments using three models, `mistral-7b-instruct-v0.2` (Jiang et al., 2023a), `llama-2-7b-chat` (Touvron et al., 2023), `llama-3-8b-instruct` (MetaAI, 2024), keeping the model used for prompt optimization, bootstrapping training traces, and fine-tuning the same in a given experiment for all modules. We initialize all the modules of an input program Φ to use the same LM weights (e.g. `mistral-7b-instruct-v0.2`), but distinct prompt templates specialized for their particular module-level task, such as generating a search query or answering a question. We implement all of our programs and optimizers as extensions to the DSPy framework.

For each dataset, we split the data into training, development, and test sets. We shuffle the training set every time before we perform a prompt or weight optimization, controlled by random seed. For prompt optimization, we sub-sample non-overlapping sets of training and validation examples from the initial training set for each task and use the `BFRS` prompt optimizer to optimize the module level prompts of a given Φ, leaving its underlying LM weights untouched. For weight optimization, we use all the available training examples as potential candidates to generate the data for fine-tuning Φ's LM weights and pass them to the `BFT` weight optimizer, which: (1) runs a given Φ on all the training examples, (2) keeps the traces where the final output of Φ was correct and filters out the rest, (3) gets module level prompt-completion pairs for each trace, (4) creates new pairs by replacing the prompt generated by Φ with a vanilla prompt, (5) combines all the module level prompt-completion pairs into one dataset, (6) fine-tunes Φ's LM weights on this dataset, and finally, (7) returns an updated Φ where the LM weights of all the modules are set to the fine-tuned LM. We use the Low Rank Adaptation (LoRA; Hu et al. 2022) method to fine-tune our LMs.

The full text for our programs and the vanilla prompts are shared in Appendix A. The license information for all LMs and datasets used as well as our implementation details such as hyperparameters and software are reported in Appendix B and Appendix C, respectively.

Multi-hop Reasoning `HotPotQA` (in the "full-wiki" setting) is a question answering task in which systems must find two Wikipedia page abstracts out of a corpus of 5 million via search and use them to answer a factoid question. Therefore it can be implemented as a program that has three LM modules: the first two for generating *search queries* (i.e., *hops*) and the last one for generating an *answer*. Each module uses Chain-of-Thought (CoT; Wei et al. 2022) to generate its outputs, producing a *reasoning* string before the search query or the answer. Search queries are passed to a frozen `ColBERTv2` (Santhanam et al., 2022) retriever. Accuracy is measured using the exact match score of the answer with the ground truth answer for the given question, after normalizing case, stripping surrounding whitespace characters, and removing punctuation. We use the following splits for `HotPotQA`: 1000 training examples and 500 development examples drawn from the original training set, along with 1500 test examples drawn from the original validation set, since the original test set is not public. We sub-sample 100 and 250 non-overlapping examples for the prompt optimization training and validation sets, respectively.

Arithmetic Reasoning `GSM8K` is a benchmark consisting of grade school math problems. We implement it as an LM program with a single module using CoT prompting, where the LM generates a reasoning string followed by an answer. We measure accuracy by extracting the last number from the first line of the model's response and comparing it to the ground truth response. For `GSM8K`, we use a training set of 1000 examples and a development set of 500 examples, both sampled from the original training set. We use all the 1319 test examples available for our test set. We use 100 and 250 non-overlapping examples for training and validation sets used during prompt optimization.

Classification `Iris` is a classic classification task, where the goal is to classify species of *Iris* flowers. We use a single-module CoT DSPy program for `Iris`, with the goal of assessing whether it being a feature-based classification task gives a large advantage to methods based entirely on gradient descent (fine-tuning). This tests the extrapolation of our hypothesis to a different setting from the other two tasks. We measure accuracy using the exact match score of the answer and the correct answer given a question after normalizing both, as is the case for `HotPotQA`. The `Iris` dataset has a total of 150 examples, from which we create training, development, and test splits of equal size. We sub-sample 15 and 35 non-overlapping examples to be used as the training and validation sets for prompt optimization, respectively.

5 Results & Discussion

Held-out test set performance of the strategies described in Section 3 is shared in Table 1. Reported values are averaged across three runs with unique random seeds. Results for the individual runs are reported separately in Appendix D.

In 7 out of the 9 dataset and LM pairs, we observe that the best-performing strategies are the strategies that utilize prompt (Π) and weight (Θ) optimization steps together, although there is no clear winner among the benchmarked **`BetterTogether`** strategies that optimize both. Overall, optimizing prompts is essential on all the tasks, but optimizing prompts and weights together leads to strong gains over the best setting that only optimizes one of the two.

In summary, we have proposed to *alternate between optimizing prompts and fine-tuning LM weights* and explored a few strategies for doing so. In experiments with multi-hop QA (`HotPotQA`), mathematical reasoning (`GSM8K`), and feature-based classification (`Iris`), we show that our strategies are highly effective for getting an LM to teach itself to perform an LM program via bootstrapping, leading to 5–78% gains for `HotPotQA`, 2.5–10% gains for `GSM8K`, and 3.5–88% gains for `Iris`.

6 Limitations

While this paper presents strong evidence from nine cases, spanning three tasks (and their corresponding LM programs) and three LMs, it is possible that other tasks, programs, or LMs will change the pattern in unforeseen ways. In particular, we have only experimented with weight optimization in the form of LoRA fine-tuning of pre-trained models. It is in principle possible that some other fine-tuning strategy would be so powerful and cost-effective as to remove the need for prompt optimization.

In addition, though we expect our findings to inform many researchers and practitioners interested in optimizing LM programs, and encourage them to explore optimizing prompts and fine-tuning LM weights together, we do not yet understand *why* both are important. The role of prompt optimization and the role of fine-tuning in multi-stage LM programs are both new, and the relative lack of understanding of these roles in the emerging literature could pose risks in unanticipated interactions between these components, compared with standard gradient descent for neural networks, which has been studied for decades.

Acknowledgments

D.S. is supported by Ravi Family Graduate Fellowship. This work was partially supported by IBM as a founding member of the Stanford Institute for Human-Centered Artificial Intelligence (HAI), and by the HAI Hoffman–Yee Grant “Dendritic Computation for Knowledge Systems”.

References

Luca Beurer-Kellner, Marc Fischer, and Martin Vechev. 2023. Prompting is programming: A query language for large language models. *Proc. ACM Program. Lang.*, 7(PLDI).

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. 2021. Training verifiers to solve math word problems. *Preprint*, arXiv:2110.14168.

David Dohan, Winnie Xu, Aitor Lewkowycz, Jacob Austin, David Bieber, Raphael Gontijo Lopes, Yuhuai Wu, Henryk Michalewski, Rif A. Saurous, Jascha Sohl-dickstein, Kevin Murphy, and Charles Sutton. 2022. Language model cascades. *Preprint*, arXiv:2207.10342.

Karel D’Oosterlinck, Omar Khattab, François Remy, Thomas Demeester, Chris Develder, and Christopher Potts. 2024. In-context learning for extreme multi-label classification. *Preprint*, arXiv:2401.12178.

Ronald A. Fisher. 1988. Iris. UCI Machine Learning Repository.

Kelvin Guu, Kenton Lee, Zora Tung, Panupong Pasupat, and Ming-Wei Chang. 2020. Retrieval augmented language model pre-training. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pages 3929–3938. PMLR.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. 2022. Lora: Low-rank adaptation of large language models. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net.

HuggingFace. 2023. Text generation inference.

Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023a. Mistral 7b. *Preprint*, arXiv:2310.06825.

Zhengbao Jiang, Frank Xu, Luyu Gao, Zhiqing Sun, Qian Liu, Jane Dwivedi-Yu, Yiming Yang, Jamie Callan, and Graham Neubig. 2023b. Active retrieval augmented generation. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 7969–7992, Singapore. Association for Computational Linguistics.

Omar Khattab, Christopher Potts, and Matei A. Zaharia. 2021. Baleen: Robust multi-hop reasoning at scale via condensed retrieval. In *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pages 27670–27682.

Omar Khattab, Keshav Santhanam, Xiang Lisa Li, David Hall, Percy Liang, Christopher Potts, and Matei Zaharia. 2022. Demonstrate-search-predict: Composing retrieval and language models for knowledge-intensive nlp. *ArXiv preprint*, abs/2212.14024.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan A, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. 2024. DSPy: Compiling declarative language model calls into state-of-the-art pipelines. In *The Twelfth International Conference on Learning Representations*.

Patrick S. H. Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, Sebastian Riedel, and Douwe Kiela. 2020. Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*.

Xinbei Ma, Yeyun Gong, Pengcheng He, Hai Zhao, and Nan Duan. 2023. Query rewriting in retrieval-augmented large language models. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 5303–5315, Singapore. Association for Computational Linguistics.

Dirk Merkel. 2014. Docker: lightweight linux containers for consistent development and deployment. *Linux J.*, 2014(239).

MetaAI. 2024. Meta llama 3.

Krista Opsahl-Ong, Michael J Ryan, Josh Purtell, David Broman, Christopher Potts, Matei Zaharia, and Omar Khattab. 2024. Optimizing instructions and demonstrations for multi-stage language model programs.

Mohammadreza Pourreza and Davood Rafiei. 2023. DIN-SQL: decomposed in-context learning of text-to-sql with self-correction. In *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*.

Ofir Press, Muru Zhang, Sewon Min, Ludwig Schmidt, Noah Smith, and Mike Lewis. 2023. Measuring and narrowing the compositionality gap in language models. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pages 5687–5711, Singapore. Association for Computational Linguistics.

Peng Qi, Haejun Lee, Tg Sido, and Christopher Manning. 2021. Answering open-domain questions of varying reasoning steps from text. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 3599–3614, Online and Punta Cana, Dominican Republic. Association for Computational Linguistics.

Keshav Santhanam, Omar Khattab, Jon Saad-Falcon, Christopher Potts, and Matei Zaharia. 2022. ColBERTv2: Effective and efficient retrieval via lightweight late interaction. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 3715–3734, Seattle, United States. Association for Computational Linguistics.

Imanol Schlag, Sainbayar Sukhbaatar, Asli Celikyilmaz, Wen tau Yih, Jason Weston, Jürgen Schmidhuber, and Xian Li. 2023. Large language model programs. *Preprint*, arXiv:2305.05364.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open foundation and fine-tuned chat models.

Vijay Viswanathan, Chenyang Zhao, Amanda Bertsch, Tongshuang Wu, and Graham Neubig. 2023. Prompt2Model: Generating deployable models from natural language instructions. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 413–421, Singapore. Association for Computational Linguistics.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Brian Ichter, Fei Xia, Ed H. Chi, Quoc V. Le, and Denny Zhou. 2022. Chain-of-thought prompting elicits reasoning in large language models. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*.

Zhilin Yang, Peng Qi, Saizheng Zhang, Yoshua Bengio, William Cohen, Ruslan Salakhutdinov, and Christopher D. Manning. 2018. HotpotQA: A dataset for diverse, explainable multi-hop question answering. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 2369–2380, Brussels, Belgium. Association for Computational Linguistics.

Eric Zelikman, Yuhuai Wu, Jesse Mu, and Noah D. Goodman. 2022. Star: Bootstrapping reasoning with reasoning. In *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*.

Appendices

A Programs

The DSPy programs for `HotPotQA`, `GSM8K`, and `Iris` are shared in Python Snippets A.1, A.2, A.3, respectively. Tables A.1, A.2, A.3 show the vanilla (zero-shot) programs generated with these programs, along with sample model completions.

```python
class HotPotQAProgram(dspy.Module):
    def __init__(self, passages_per_hop=3):
        super().__init__()

        self.retrieve = dspy.Retrieve(k=passages_per_hop)
        self.generate_query = [dspy.ChainOfThought("context, question ->
            search_query") for _ in range(2)]
        self.generate_answer = dspy.ChainOfThought("context, question -> answer")

    def forward(self, question):
        context = []

        for hop in range(2):
            search_query = self.generate_query[hop](context=context, question=
                question).search_query
            passages = self.retrieve(search_query).passages
            context = dsp.utils.deduplicate(context + passages)

        return self.generate_answer(context=context, question=question).copy(
            context=context)
```

Python Snippet A.1: DSPy program for `HotPotQA`, consisting of three Chain-of-Thought (CoT) LM modules, along with a frozen retriever. Each LM module in the program uses its own prompt and LM weights under the hood.

Table A.1: Vanilla (zero-shot) prompts for the `HotPotQA` program given in Python Snippet A.1 with three LM modules, each with corresponding prompts and LM weights. **(1)** the `generate_query[0]` module prompts the corresponding LM to generate a search query given a question along with an empty context; **(2)** the `generate_query[1]` module prompts the corresponding model to generate a search query given the context paragraphs retrieved using the search query outputted by the prior module; **(3)** the `generate_answer` module prompts the corresponding LM to answer the question given the context paragraphs retrieved by the earlier modules. The green text shows the model completions obtained by setting all the modules of the program to use `mistralai/Mistral-7b-Instruct-v0.2` as their corresponding LM. The correct answer to the question is indeed `Key deer` as predicted.

Module Label	Prompt
`generate_query[0]`	Given the fields \`context\`, \`question\`, produce the fields \`search_query\`. --- Follow the following format. Context: ${context} Question: ${question} Reasoning: Let's think step by step in order to ${produce the search_query}. We ... Search Query: ${search_query} --- Context: N/A Question: What animal subject to the Foster's rule is the smallest North American deer? Reasoning: Let's think step by step in order to find the smallest North American deer subject to the Foster's rule. 1. The Foster's rule is a mathematical formula used to estimate the body weight of large mammals based on their ear surface area. 2. To apply the Foster's rule, we need to find the ear surface area of the deer species in question. 3. We don't have the ear surface area of each North American deer species in the context, so we need to search for it. 4. Therefore, our search query should include the keywords "North American deer," "ear surface area," and "Foster's rule." Search Query: North American deer ear surface area Foster's rule smallest species

Module Label	Prompt
generate_query[1]	

```
Given the fields `context`, `question`, produce the fields `search_query`.

---

Follow the following format.

Context: ${context}

Question: ${question}

Reasoning: Let's think step by step in order to ${produce the
search_query}. We ...

Search Query: ${search_query}

---

Context:
[1] «Key deer | The Key deer (”Odocoileus virginianus clavium”) is an
endangered deer that lives only in the Florida Keys. It is a subspecies of
the white-tailed deer (”O. virginianus”). It is the smallest North American
deer.»
[2] «North American least shrew | The North American least shrew
(”Cryptotis parva”) is one of the smallest mammals, growing to be only up
to 3 inches long. The North American least shrew has a long pointed snout
and a tail never more than twice the length of its hind foot. It has a
dense fur coat that is either grayish-brown or reddish-brown with a white
belly. Its fur becomes lighter in the summer and darker in the winter.
Although similar in appearance to several species of rodents, all shrews
are members of the order Soricomorpha and should not be mistaken for a
member of the Rodentia order. The North American least shrew's eyes are
small and its ears are completely concealed within its short fur, giving it
very poor eyesight and hearing.»
[3] «Wynnea americana | Wynnea americana, commonly known as moose antlers
or rabbit ears, is a species of fungus in the family Sarcoscyphaceae. This
uncommon inedible species is recognizable by its spoon-shaped or rabbit-ear
shaped fruit bodies that may reach up to 13 cm tall. It has dark brown and
warty outer surfaces, while the fertile spore-bearing inner surface is
orange to pinkish to reddish brown. The fruit bodies grow clustered
together from large underground masses of compacted mycelia known as
sclerotia. In eastern North America, where it is typically found growing in
the soil underneath hardwood trees, it is found from New York to Michigan
south to Mexico. The species has also been collected from Costa Rica,
India, and Japan.»

Question: What animal subject to the Foster's rule is the smallest North
American deer?

Reasoning: Let's think step by step in order to produce the search query.
We need to find the smallest North American deer, which is mentioned in the
context as the Key deer. Therefore, the search query should include the
terms ”Key deer” and ”smallest North American deer”.

Search Query: Key deer smallest North American deer
```

Module Label	Prompt

generate_answer

```
Given the fields `context`, `question`, produce the fields `answer`.

---

Follow the following format.

Context: ${context}

Question: ${question}

Reasoning: Let's think step by step in order to ${produce the answer}. We
...

Answer: ${answer}

---

Context:
[1] «Key deer | The Key deer (”Odocoileus virginianus clavium”) is an
endangered deer that lives only in the Florida Keys. It is a subspecies of
the white-tailed deer (”O. virginianus”). It is the smallest North American
deer.»
[2] «North American least shrew | The North American least shrew
(”Cryptotis parva”) is one of the smallest mammals, growing to be only up
to 3 inches long. The North American least shrew has a long pointed snout
and a tail never more than twice the length of its hind foot. It has a
dense fur coat that is either grayish-brown or reddish-brown with a white
belly. Its fur becomes lighter in the summer and darker in the winter.
Although similar in appearance to several species of rodents, all shrews
are members of the order Soricomorpha and should not be mistaken for a
member of the Rodentia order. The North American least shrew's eyes are
small and its ears are completely concealed within its short fur, giving it
very poor eyesight and hearing.»
[3] «Wynnea americana | Wynnea americana, commonly known as moose antlers
or rabbit ears, is a species of fungus in the family Sarcoscyphaceae. This
uncommon inedible species is recognizable by its spoon-shaped or rabbit-ear
shaped fruit bodies that may reach up to 13 cm tall. It has dark brown and
warty outer surfaces, while the fertile spore-bearing inner surface is
orange to pinkish to reddish brown. The fruit bodies grow clustered
together from large underground masses of compacted mycelia known as
sclerotia. In eastern North America, where it is typically found growing in
the soil underneath hardwood trees, it is found from New York to Michigan
south to Mexico. The species has also been collected from Costa Rica,
India, and Japan.»
[4] «Pudú | The pudús (Mapudungun ”püdü” or ”püdu”, Spanish: pudú , ] ) are
two species of South American deer from the genus Pudu, and are the world's
smallest deer. The name is a loanword from Mapudungun, the language of the
indigenous Mapuche people of southern Chile and south-western Argentina.
The two species of pudús are the northern pudú (”Pudu mephistophiles”) from
Venezuela, Colombia, Ecuador, and Peru, and the southern pudú (”Pudu puda”;
sometimes incorrectly modified to ”Pudu pudu”) from southern Chile and
south-western Argentina. Pudús range in size from 32 to tall, and up to 85
cm long. As of 2009, the southern pudu is classified as near threatened,
while the northern pudu is classified as vulnerable in the IUCN Red List.»

Question: What animal subject to the Foster's rule is the smallest North
American deer?

Reasoning: Let's think step by step in order to find the answer. We know
that the smallest North American deer is mentioned in the context. We also
know that the Foster's rule refers to a group of animals. In this case, the
Foster's rule refers to endangered deer species that live only in the
Florida Keys. Therefore, the Key deer is the answer.

Answer: Key deer
```

```python
class CoTProgram(dspy.Module):
    def __init__(self):
        super().__init__()
        self.generate_answer = dspy.ChainOfThought("question -> answer")

    def forward(self, question):
        return self.generate_answer(question=question)
```

Python Snippet A.2: DSPy program for `GSM8K`.

Table A.2: Vanilla (zero-shot) prompts for the `GSM8K` program given in Python Snippet A.2 with one LM module. The `generate_answer` module prompts the corresponding LM to answer a grade school math problem. The green text shows the model completion obtained by setting all the modules of the program to use `mistralai/Mistral-7b-Instruct-v0.2` as their corresponding LM. The model's response is evaluated as correct because the correct answer (`1200`) is contained in the first line of model's generation.

Module Label	Prompt
`generate_answer`	

```
Given the fields `question`, produce the fields `answer`.

---

Follow the following format.

Question: ${question}
Reasoning: Let's think step by step in order to ${produce the answer}. We
...
Answer: ${answer}

---

Question: Micah can type 20 words per minute and Isaiah can type 40 words
per minute. How many more words can Isaiah type than Micah in an hour?
Reasoning: Let's think step by step in order to find out how many more
words Isaiah can type than Micah in an hour. We know that Micah can type 20
words per minute and Isaiah can type 40 words per minute. In one minute,
Isaiah types twice as many words as Micah. In 60 minutes, Isaiah types 60
minutes * 40 words per minute = <<60*40=2400>>2400 words. Micah types 60
minutes * 20 words per minute = <<60*20=1200>>1200 words. The difference
between the number of words Isaiah and Micah can type in an hour is 2400
words - 1200 words = <<2400-1200=1200>>1200 words.

Answer: Isaiah can type 1200 more words than Micah in an hour.
```

```python
class IrisSignature(dspy.Signature):
    "Given the petal and sepal dimensions in cm, predict the iris species."

    petal_length = dspy.InputField()
    petal_width = dspy.InputField()
    sepal_length = dspy.InputField()
    sepal_width = dspy.InputField()
    answer = dspy.OutputField(desc='setosa, versicolor, or virginica')


class IrisProgram(dspy.Module):
    def __init__(self):
        self.generate_answer = dspy.ChainOfThought(IrisSignature)

    def forward(self, petal_length, petal_width, sepal_length, sepal_width):
        return self.generate_answer(petal_length=petal_length, petal_width=
            petal_width, sepal_length=sepal_length, sepal_width=sepal_width)
```

Python Snippet A.3: DSPy program for `Iris`, provided to us by the DSPy team.

Table A.3: Vanilla (zero-shot) prompts for the `Iris` program given in Python Snippet A.3 with one LM module. The `generate_answer` module prompts the corresponding LM to classify the *Iris* flowers into one of the following species given their features: *Setosa*, *Versicolor*, *Virginica*. The green text shows the model completion obtained by setting all the modules of the program to use `mistralai/Mistral-7b-Instruct-v0.2` as their corresponding LM. The correct answer is `Setosa`, just as predicted.

Module Label	Prompt
`generate_answer`	

```
Given the petal and sepal dimensions in cm, predict the iris species.

---

Follow the following format.

Petal Length: ${petal_length}

Petal Width: ${petal_width}

Sepal Length: ${sepal_length}

Sepal Width: ${sepal_width}

Reasoning: Let's think step by step in order to ${produce the answer}. We
...

Answer: setosa, versicolor, or virginica

---

Petal Length: 1.4

Petal Width: 0.2

Sepal Length: 5.0

Sepal Width: 3.6

Reasoning: Let's think step by step in order to predict the iris species.
We will compare the given measurements with the average measurements of
each iris species. 1. Setosa: The average petal length for setosa is 1.3 cm
and the average petal width is 0.3 cm. The given petal length (1.4 cm) is
slightly larger than the average, but the petal width (0.2 cm) is smaller
than the average. However, the sepal dimensions (5.0 cm and 3.6 cm) are
within the range of setosa. Based on these measurements, it is likely that
the iris is setosa. 2. Versicolour: The average petal length for
versicolour is 4.2 cm and the average petal width is 1.4 cm. The given
petal length (1.4 cm) is smaller than the average, and the petal width (0.2
cm) is much smaller than the average. The sepal dimensions (5.0 cm and 3.6
cm) are also within the range of versicolour. However, the small petal
dimensions suggest that it is less likely to be versicolour. 3. Virginica:
The average petal length for virginica is 5.8 cm and the average petal
width is 2.0 cm. The given petal length (1.4 cm) is much smaller than the
average, and the petal width (0.2 cm) is much smaller than the average. The
sepal dimensions (5.0 cm and 3.6 cm) are also within the range of
virginica. However, the small petal dimensions suggest that it is less
likely to be virginica. Based on the given measurements, it is most likely
that the iris is setosa.

Answer: setosa.
```

B Asset Information

We share the associated licenses for the models and datasets we used below. For models, we list the specific `HuggingFace` model id we used to retrieve the respective weights.

1. `mistralai/Mistral-7b-Instruct-v0.2`: `Apache License 2.0`

2. `meta-llama/Llama-2-7b-chat-hf`: `Meta Llama 2 Community License` at `https://ai.meta.com/llama/license/`

3. `meta-llama/Meta-Llama-3-8B-Instruct`: `Meta Llama 3 Community License` at `https://llama.meta.com/llama3/license/`

4. `HotPotQA`: `Apache License 2.0`

5. `GSM8K`: `MIT License`

6. `Iris`: `Creative Commons Attribution 4.0 International (CC BY 4.0)`

All the LMs used in this work are intended for use in English.

C Implementation Details

In this section, we share the implementation details as it pertains to the LM inference and fine-tuning infrastructure, as well as compute requirements.

Inference For inference, we host our models in `Docker` (Merkel, 2014) instances through `HuggingFace's text-generation-inference` (HuggingFace, 2023) toolkit. We keep the sampling parameters the same across all experiments, using `TopK` sampling with a `temperature` of 0.1, and `top_k` of 0.97, until the model either generates a stopping string or a total of 1024 tokens including the tokens in the prompt.

Prompt Optimization For prompt optimization, we use the `BootstrapFewShotRS` (`BFRS`) optimizer from the DSPy library. In particular, we allow `BFRS` to randomly search 6 candidate programs using up to 3 few-shot examples for each module prompt.

Fine-tuning For fine-tuning, we use Low Rank Adaptation (LoRA; Hu et al. 2022) to train the `query` and `key` self-attention layers of our models, using a LoRA `rank` of 32, `alpha` of 64, with no dropout. We fine-tune all of our models for 5 epochs using `bfloat16` precision, with a learning rate of $1\text{e}{-5}$ and an *effective batch size* of 8.Compute The **`BetterTogether`** strategies explored in this paper are naturally more expensive to run when compared to just prompt optimizing or fine-tuning. How these two steps compare to each other in terms of compute requirements or wall clock time depends on the particular settings used for each as well as the size of the dataset used. Total approximate GPU hours to produce Table 1 was $\approx$75 hours, using `A100` GPUs.

D Extended Results

The results shared in Table 1 are the average of three runs. Tables D.1, D.2, and D.3 show the breakdown of the individual runs for `HotPotQA`, `GSM8K`, and `Iris`, respectively.

Strategy	`mistral-7b-instruct-v0.2`				`llama-2-7b-chat`				`llama-3-8b-instruct`			
	Run 1	Run 2	Run 3	Avg	Run 1	Run 2	Run 3	Avg	Run 1	Run 2	Run 3	Avg
Baseline Strategies												
Vanilla Zero-shot	17.2	17.2	17.2	17.2	13.2	13.2	13.2	13.2	31.6	31.6	31.6	31.6
Prompt Optimization (Π)	32.7	34.7	34.0	33.8	33.3	33.3	33.4	33.3	45.7	47.4	47.5	**46.9**
Weight Optimization (Θ)	22.0	23.1	23.5	22.9	12.4	11.8	12.3	12.2	34.9	35.3	34.3	34.8
$\Pi \rightarrow \Pi$	31.7	36.0	33.7	33.8	31.7	33.1	33.1	32.6	47.3	45.4	46.7	46.5
$\Theta \rightarrow \Theta$	24.1	23.9	23.9	24.0	12.4	13.5	13.3	13.0	35.1	34.1	34.1	34.4
BetterTogether Strategies												
$\Pi \rightarrow \Theta$	34.9	39.1	34.9	36.3	32.8	32.3	33.1	32.7	40.6	42.1	45.7	42.8
$\Theta \rightarrow \Pi$	29.3	33.8	35.8	33.0	36.0	33.4	33.1	34.2	44.5	40.9	45.3	43.6
$\Pi \rightarrow \Theta \rightarrow \Pi$	34.9	40.7	37.2	**37.6**	34.7	34.5	35.3	**34.8**	46.5	47.1	46.4	46.7

Table D.1: Results of `HotPotQA` Runs. Percentage accuracies of baseline and **`BetterTogether`** strategies on `HotPotQA` evaluated on `mistral-7b-instruct-v0.2`, `llama-2-7b-chat`, and `llama-3-8b-instruct`. Reported are performance of 3 runs on a held-out test set of 1500 examples, using different random seeds. **Bold** font in the average columns (Avg) is used to mark the highest score in a given column.

Strategy	`mistral-7b-instruct-v0.2`				`llama-2-7b-chat`				`llama-3-8b-instruct`			
	Run 1	Run 2	Run 3	Avg	Run 1	Run 2	Run 3	Avg	Run 1	Run 2	Run 3	Avg
Baseline Strategies												
Vanilla Zero-shot	40.3	40.3	40.3	40.3	24.0	24.0	24.0	24.0	72.7	72.7	72.7	72.7
Prompt Optimization (Π)	45.0	47.2	47.1	46.4	27.3	25.1	25.5	26.0	76.9	77.9	78.9	77.9
Weight Optimization (Θ)	40.8	40.0	41.2	40.7	23.7	24.2	24.0	24.0	75.7	74.8	74.8	75.1
$\Pi \rightarrow \Pi$	46.3	47.2	49.6	47.7	28.4	24.0	21.8	24.7	76.5	80.1	76.1	77.6
$\Theta \rightarrow \Theta$	42.9	41.8	43.8	42.8	24.0	24.3	24.0	24.1	52.2	36.6	43.4	44.0
BetterTogether Strategies												
$\Pi \rightarrow \Theta$	46.4	47.3	48.2	47.3	27.8	28.1	25.9	**27.3**	77.6	75.4	79.8	77.6
$\Theta \rightarrow \Pi$	50.1	46.0	48.8	**48.3**	26.8	26.1	27.0	26.6	78.5	79.8	78.4	**78.9**
$\Pi \rightarrow \Theta \rightarrow \Pi$	44.9	48.5	47.1	46.8	27.1	25.9	25.9	26.3	77.6	75.4	77.8	77.0

Table D.2: Results of `GSM8K` Runs. Percentage accuracies of baseline and **`BetterTogether`** strategies on `GSM8K` evaluated on `mistral-7b-instruct-v0.2`, `llama-2-7b-chat`, and `llama-3-8b-instruct`. Reported are performance of 3 runs on a held-out test set of 1319 examples, using different random seeds. **Bold** font in the average columns (Avg) is used to mark the highest score in a given column.

Strategy	`mistral-7b-instruct-v0.2`				`llama-2-7b-chat`				`llama-3-8b-instruct`			
	Run 1	Run 2	Run 3	Avg	Run 1	Run 2	Run 3	Avg	Run 1	Run 2	Run 3	Avg
Baseline Strategies												
Vanilla Zero-shot	26.0	26.0	26.0	26.0	0.0	0.0	0.0	0.0	48.0	48.0	48.0	48.0
Prompt Optimization (Π)	52.0	54.0	66.0	57.3	44.0	68.0	58.0	56.7	62.0	96.0	80.0	79.3
Weight Optimization (Θ)	24.0	34.0	30.0	29.3	–	–	–	–	38.0	40.0	34.0	37.3
$\Pi \rightarrow \Pi$	48.0	64.0	66.0	59.3	66.0	70.0	56.0	64.0	70.0	94.0	82.0	**82.0**
$\Theta \rightarrow \Theta$	40.0	36.0	38.0	38.0	–	–	–	–	44.0	36.0	38.0	39.3
BetterTogether Strategies												
$\Pi \rightarrow \Theta$	32.0	26.0	34.0	30.7	30.0	26.0	24.0	26.7	50.0	42.0	40.0	44.0
$\Theta \rightarrow \Pi$	80.0	54.0	66.0	**66.7**	–	–	–	–	78.0	78.0	80.0	78.7
$\Pi \rightarrow \Theta \rightarrow \Pi$	52.0	44.0	62.0	52.7	62.0	70.0	64.0	**65.3**	74.0	80.0	84.0	79.3

Table D.3: Results of `Iris` Runs. Percentage accuracies of baseline and **`BetterTogether`** strategies on `Iris` evaluated on `mistral-7b-instruct-v0.2`, `llama-2-7b-chat`, and `llama-3-8b-instruct`. Reported are performance of 3 runs on a held-out test set of 50 examples, using different random seeds. **Bold** font in the average columns (Avg) is used to mark the highest score in a given column. Strategies where weight optimization is the first step use the vanilla (zero-shot) strategy to generate the initial fine-tuning dataset. If a model generates very few or no correct outputs under the vanilla strategy on the training set used to bootstrap the fine-tuning data, there will not be a sufficient dataset for fine-tuning. These settings are marked with "–".