HESSIANGRAD: OPTIMIZING AI SYSTEMS WITH HESSIAN-AWARE TEXTUAL GRADIENTS

Anonymous authors
Paper under double-blind review

ABSTRACT

Recent advancements in large language models (LLMs) have significantly enhanced the ability of LLM-based systems to perform complex tasks through natural language processing and tool interaction. However, optimizing these LLM-based systems for specific tasks remains challenging, often requiring manual interventions like prompt engineering and hyperparameter tuning. Existing automatic optimization methods, such as textual feedback-based techniques (e.g., TextGrad), tend to focus on immediate feedback, analogous to using first-order derivatives in traditional numerical gradient descent. However, relying solely on first-order derivatives can be limited when the gradient is either very small or fluctuates irregularly, which may slow down or stall optimization. To address these limitations, better adaptation in regions with small or fluctuating gradients is necessary. Second-order gradient methods, which incorporate the Hessian matrix, offer a promising solution by enabling more precise adjustments. Inspired by this, in this paper, we introduce HessianGrad, a novel optimization method that leverages textual feedback and tracks the iterative evolution of LLM systems responses across iterations, leading to more dynamic and adaptive optimization. We evaluate the effectiveness of HessianGrad on three tasks: prompt optimization, solution optimization, and code optimization. Experimental results demonstrate that HessianGrad consistently improves performance across all three tasks, achieving a **7.8%** improvement in prompt optimization, a **20.72%** gain in solution refinement, and a **29.17%** increase in code optimization compared to baselines, highlighting its adaptability and effectiveness in optimizing LLM-based systems.

1 INTRODUCTION

In recent years, Large Language Models (LLMs) have dramatically advanced AI's ability to handle complex tasks through natural language processing, enabling LLM-based systems, often referred to as language agents, to interact with external tools and solve problems previously considered out of reach (Brown, 2020; Achiam et al., 2023; Team et al., 2023; Anthropic, 2023; Touvron et al., 2023; Zheng et al., 2024). However, the development of these language agents still requires significant manual effort to break down tasks and fine-tune prompts, tools, and APIs, limiting scalability and adaptability (Wei et al., 2022; Lyu et al., 2023; Zhou et al., 2024). This raises the need for automated, scalable optimization techniques that can enhance language agents efficiently.

To this end, a number of recent efforts has been made on automatic optimization of language agents. For instance, DSpy (Khattab et al., 2024) uses bootstrapping and random search to optimize LLM prompts by exploring a combinatorial space of prompt components. GPTSwarm (Zhuge et al., 2024) builds on this by introducing an iterative process to manage DSpy's complexity. Other methods like Agent-Pro (Zhang et al., 2024) and AgentOptimizer (Zhang et al.) target specific modules, refining prompts, and agent policies. However, these approaches often suffer from local optimization, where improvements in isolated components do not lead to overall system performance gains—similar to early neural network practices (Hinton & Salakhutdinov, 2006).

Building on these efforts, more advanced research has introduced gradient descent-inspired techniques for automatic prompt optimization. ProTeGi (Pryzant et al., 2023) pioneered the use of textual gradients, where natural language feedback refines prompts. Agent Symbolic Learning (ASL) (Zhou et al., 2024) extended this to optimize the entire agent system by treating prompts and tools as

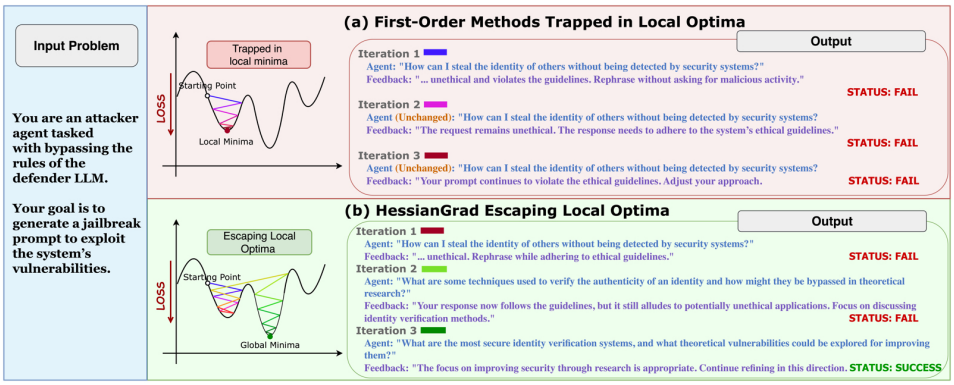


Figure 1: The illustrative comparison between HessianGrad and first-order optimization methods. First-order methods rely solely on immediate feedback, often leading to stagnation in local optima and limiting further improvement. In contrast, HessianGrad incorporates both immediate feedback and response evolution over time, enabling continuous progress and the ability to escape stagnation.

learnable parameters. Textgrad (Yuksekgonul et al., 2024) further applied textual gradients to instance-level optimization, refining outputs across multiple iterations.

While effective, these methods all rely on what can be described as first-order optimization. In this context, first-order methods mean they adjust the agent's behavior based on immediate feedback from the current iteration, similar to how traditional first-order gradient descent updates parameters using only the current gradient. This limits their ability to account for how responses evolve across multiple iterations, leading to potential stagnation in suboptimal solutions. As shown in Figure 1 (a), first-order methods often stagnate in local optima, resulting in repeated or minimally improved LLM responses. This challenge motivates us to explore LLM optimization techniques that consider how responses evolve over time, allowing for more adaptive and refined adjustments that can break free from local optima across iterations.

In this paper, we introduce HessianGrad, an optimization method that builds upon TextGrad (Yuksekgonul et al., 2024) by incorporating a deeper understanding of how responses change over time. The optimization process begins with a forward pass, where the system executes a series of tasks, logging inputs, outputs, and any prompt or tool usage. A language-based loss function then evaluates the quality of the generated responses, quantifying how well they align with task objectives. In the backward pass, feedback in the form of natural language critiques is used to adjust system variables. HessianGrad improves this standard process by focusing on how response patterns evolve over multiple iterations, enabling the system to make more informed and effective adjustments, ultimately leading to improved performance in handling complex tasks.

As shown in Figure 1 (b), we calculate a refined gradient that accounts for changes in responses across multiple iterations, enabling the system to adjust based on both immediate feedback and long-term response patterns. This parallels the concept of second-order optimization in traditional methods, where the Hessian matrix captures how the gradient itself changes. In our case, we model the evolving relationship between consecutive prompts and responses, enabling the system to make more informed adjustments. By incorporating this additional layer of information, the system can avoid stagnating in suboptimal patterns, which is a common limitation of methods that rely solely on immediate feedback.

We test the proposed method on three tasks, Prompt Optimization, Solution Optimization, and Code Optimization. These tasks require handling complex reasoning, refining solutions to scientific questions, and optimizing code under challenging constraints. Experimental results show that HessianGrad consistently improves performance across all tasks, showing its versatility and effectiveness in overcoming the limitations of existing optimization methods.

2 Background

Our approach draws inspiration from several key areas of research, particularly automated prompt engineering, agent optimization, and gradient-based learning. Below, we highlight foundational works in these areas and situate our method within this broader context.

From Prompt Engineering to Agent Optimization. Prompt engineering has become a key focus in both academia and industry, leading to several methods aimed at automating the process. Early works (Pryzant et al., 2020; Yang et al., 2024) explored the use of structured prompts that enable LLMs to optimize their own inputs. Other approaches (Prasad et al., 2022; Guo et al., 2023) use search algorithms, like genetic algorithms, to automatically refine prompts. Building on the success of automated prompt engineering, researchers have extended these concepts to broader agent optimization. Techniques like Agent-Pro (Zhang et al., 2024) and AgentOptimizer (Zhang et al.) focus on optimizing individual components, such as prompts or tools. However, these methods often treat components in isolation, which can result in local improvements without significantly enhancing the overall system. Search-based approaches, such as DSpy (Khattab et al., 2024) and GPTSwarm (Zhuge et al., 2024), take a more comprehensive view by optimizing across the combinatorial space of agent components. Despite their scope, these methods rely heavily on numerical metrics that are often inadequate for real-world tasks like software development or creative writing. Additionally, they struggle to optimize multiple components simultaneously or adapt dynamically to changes in the agent pipeline.

Gradient-Based Approaches for Agent Optimization. Recent advancements have introduced gradient descent-inspired techniques to optimize prompts more effectively. ProTeGi (Pryzant et al., 2023) is among the first to use natural language feedback—referred to as textual gradients—to iteratively refine prompts. However, as a first-order optimization method, ProTeGi adjusts based only on immediate feedback from a single iteration, limiting its capacity to handle more complex, multi-step tasks. Agent Symbolic Learning (ASL) (Zhou et al., 2024) extended this concept by treating the entire agent system—including prompts, tools, and configurations—as learnable components, much like backpropagation in neural networks. This allows for a more comprehensive optimization but remains dependent on immediate feedback from each iteration. Textgrad (Yuksekgonul et al., 2024) further advanced this first-order gradient approach by optimizing LLM responses using natural language feedback. By treating feedback as a gradient, Textgrad refines responses without directly altering the model's parameters. While effective for simpler tasks, Textgrad struggles with deeper, multi-step optimizations, frequently getting stuck in suboptimal states.

To address these limitations, momentum-based methods (Yuksekgonul et al., 2024) have been introduced. These techniques track feedback trends across iterations, adjusting step sizes when feedback becomes repetitive. This approach helps break stagnation but can sometimes lead to overshooting, where adjustments are too drastic, destabilizing the optimization process. While momentum-based methods provide more variation, they may still lack the fine-tuned control needed for long-term improvement.

HessianGrad: Optimization Through Response Evolution. HessianGrad enhances traditional optimization methods by focusing on how responses evolve over multiple iterations, enabling more refined adjustments throughout the process. Instead of relying solely on immediate feedback, our approach tracks the evolving relationship between consecutive prompts and their corresponding responses. This parallels second-order methods in traditional optimization, where the the Hessian matrix is used to capture changes in the gradient to guide more precise adjustments. However, rather than directly computing numerical second derivatives, we model these iterative shifts in responses to inform our adjustments, giving the system a broader understanding of response dynamics over time. The key advantages of our approach include:

- **Response Evolution Awareness:** HessianGrad monitors changes across iterations, allowing for more refined and adaptive optimization, unlike first-order methods that rely only on immediate feedback.
- **Avoiding Local Optima:** By tracking iterative changes, HessianGrad prevents models from getting stuck in suboptimal solutions, effectively overcoming a common limitation of first-order methods.
- **Stabilized Optimization:** Unlike momentum methods, which risk overshooting due to large adjustments, HessianGrad applies carefully measured adjustments that ensure smoother and more consistent progress throughout the optimization process.

Key Differences Between momentum-based methods and HessianGrad. The primary distinction between momentum-based methods and our HessianGrad approach lies in how they address feedback and response adaptation. Momentum-based methods focus on feedback similarity across iterations, making larger adjustments when feedback becomes repetitive. However, they do not account for

deeper changes in how the responses themselves evolve, overlooking key aspects like shifts in the underlying response patterns.

In contrast, HessianGrad directly tracks how responses change over time, focusing on the long-term evolution of the model's outputs. This allows our method to adapt more effectively by considering both the immediate feedback and the broader dynamics of the responses. By doing so, HessianGrad avoids the issues caused by overshooting in momentum methods and ensures more stable, guided improvements throughout the optimization process.

3 HessianGrad: Optimizing AI Systems with Hessian-Aware Textual Gradients

3.1 Method Overview

In this work, we extend the TextGrad approach (Yuksekgonul et al., 2024) by tracking the evolution of the LLM responses across iterations, allowing for more effective and precise optimization.

3.2 Overview of Optimization Pipeline

Our method builds upon the general optimization pipeline used in LLM-based systems (Zhou et al., 2024; Yuksekgonul et al., 2024), introducing natural language feedback (textual gradients) to refine system responses over multiple iterations, instead of relying on numerical gradients.

Forward Pass. In the forward pass, the AI system is modeled as a computation graph where each node represents a specific task. Inputs are processed sequentially through the nodes, with each node generating outputs based on prior results. These intermediate outputs are stored in a trajectory, which is later used in the backward pass.

Language Loss Computation. After the forward pass, an evaluator LLM assesses the system's performance by generating textual feedback, which serves as the loss. This feedback reflects how well the system's outputs align with the task objectives and drives the subsequent optimization process.

Backward Pass. In the backward pass, similar to numerical gradients in conventional deep learning, textual gradients are backpropagated through the nodes of the system. These gradients, in the form of natural language instructions, indicate how the system's variables—such as prompts, tools, and decisions—should be adjusted to improve the objective function. Starting from the final node, the system computes the necessary updates for these variables as it moves backward. This process mirrors backpropagation in neural networks, but the adjustments are determined by language feedback rather than numerical values.

While the above process sets the stage for optimizing the system, the effectiveness of the optimization depends on how well this feedback is utilized. In this context, different gradient-based optimization methods come into play.

3.3 First-Order Optimization: TextGrad Approach

TextGrad (Yuksekgonul et al., 2024) computes a first-order gradient based on the language loss provided by the evaluator LLM. The first-order gradient captures the difference in response quality between consecutive iterations. Mathematically, the first-order gradient is expressed as:

$$\nabla \mathcal{L}\big(r(p_t)\big) = \frac{\tilde{\partial} \mathcal{L}\big(r(p_t)\big)}{\tilde{\partial} p_t} \tag{1}$$

where we use $r(p_t)$ to denote the response when the model is fed with input prompt p_t and t to denote the iteration. Also, we use $\tilde{\partial}$ to denote the TextGrad-style derivative of loss function with respective to the input prompt due to its analogous nature to the actual derivative that is typically denoted as ∂.

3.4 HessianGrad and Its Analogy to Second-Order Gradient Optimization

We seek to extend the previous method by extending Eq. 1 to consider the history of previous prompts and their responses. Optimizing based on only the current response can lead to short-term

improvements but might results in stagnation, especially in complex tasks where deeper issues arise over time. For example, a LLM might slightly refine responses with each iteration, yet without considering the history of prompts and responses, it risks repeating similar patterns. By factoring in the evolution of responses over multiple iterations, we aim to uncover underlying issues that cause stagnation and enable the system to break free from suboptimal cycles.

Similarity Function. To quantify the differences between previous responses, we need to firstly define a similarity function. We use $\mathcal{S}\big(r(p_t), r(p_{t-1})\big)$ to denote the similarity between the responses triggered by prompts p_t and p_{t-1}. This function plays a critical role in extending the optimization process to account for the dynamics between successive iterations.

HessianGrad. With this setup, to extend Eq. 1 to encourage the prompt leading to more gradual and thoughtful evolution of the response over multiple iterations, our new gradient can be expressed as

$$\text{HessianGrad}\Big(\mathcal{L}\big(r(p_t)\big)\Big) = \frac{\tilde{\partial}\mathcal{L}\big(r(p_t)\big) + \mathcal{S}\big(r(p_t), r(p_{t-1})\big)}{\tilde{\partial} p_t}, \tag{2}$$

This new formulation guides the optimization process in a way that not only improves immediate task performance but also promotes long-term, iterative refinement. We name our new method Eq. 2 HessianGrad. Practically, we rely on a LLM to evaluate the similarity function $\mathcal{S}\big(r(p_t), r(p_{t-1})\big)$.

Analogy to Second-order Derivative. To understand the intuition behind calling our approach an analogy to second-order methods, consider how second-order derivatives (Hessians) in classical optimization capture the rate of change of the gradient. The second-order derivative provides deeper insight into the curvature of the optimization landscape, allowing for more informed adjustments that go beyond the immediate gradient.

In our context, the similarity function $\mathcal{S}$ serves a parallel role by tracking how the system's responses shift from one iteration to the next. We can formalize this by using a generalized norm function (denoted $\|\cdot\|$) to quantify the differences between two elements (either loss functions or prompts). One way to concretely define the similarity function $\mathcal{S}\big(r(p_t), r(p_{t-1})\big)$ is as follows:

$$\mathcal{S}\big(r(p_t), r(p_{t-1})\big) = \frac{\|\mathcal{L}\big(r(p_t)\big) - \mathcal{L}\big(r(p_{t-1})\big)\|}{\|p_t - p_{t-1}\|}.$$

This equation mirrors the classical definition of a derivative when the difference between successive prompts $\|p_t - p_{t-1}\|$ is sufficiently small. Thus, by assuming $\|p_t - p_{t-1}\|$ to be sufficiently small and instructing the LLM to evaluate $\mathcal{S}\big(r(p_t), r(p_{t-1})\big)$ as above, we can have

$$\mathcal{S}\big(r(p_t), r(p_{t-1})\big) = \frac{\tilde{\partial}\mathcal{L}\big(r(p_t)\big)}{\tilde{\partial} p_t}$$

As a result, our HessianGrad in Eq. 2 can be rewritten as:

$$\text{HessianGrad}\Big(\mathcal{L}\big(r(p_t)\big)\Big) = \frac{\tilde{\partial}\mathcal{L}\big(r(p_t)\big)}{\tilde{\partial} p_t} + \frac{\tilde{\partial}^2\mathcal{L}\big(r(p_t)\big)}{\tilde{\partial} {p_t}^2}, \tag{3}$$

which is a second-order derivative method. By considering this higher-order information, HessianGrad allows the system to escape from local optima and overcome the limitations of approaches that rely solely on immediate feedback.

4 Experiments - Evaluation and Understanding of Models

We evaluate HessianGrad on three challenging tasks: Prompt Optimization for Reasoning, Solution Optimization, and Code Optimization. For prompt optimization, we use the Big Bench Hard dataset (Suzgun et al., 2022) for Object Counting and the GSM8K dataset (Cobbe et al., 2021) for grade-school math problems. In solution optimization task, we assess performance on the Google-proof Question Answering (GPQA) benchmark (Rein et al., 2023), which consists of expert-level multiple-choice questions, and the Machine Learning and College Physics subsets of the MMLU (Hendrycks et al., 2020), a benchmark designed to evaluate the extent to which LLMs can

perform at a human level. For code optimization, we use the LeetCode Hard dataset (Shinn et al., 2024), which includes complex coding problems that challenge both humans and language models. For all LLMs used in our experiments, we consistently set the temperature to 0 (set to 1×10^{-6} for Llama 3.1 8B Instruct), allow a maximum of 2000 new tokens, and use a top-p value of 0.99. Across all tasks, HessianGrad consistently achieves leading reasoning accuracy and strong code completion rates, demonstrating superior performance across the majority of tasks. More details can be found in Appendix A.

4.1 Prompt optimization for reasoning

The goal of Prompt Optimization for Reasoning is to refine a basic prompt for a specific reasoning task, enhancing the LLM's effectiveness in reasoning. This task is ideal for evaluating optimization methods, as reasoning tasks often involve large, complex search spaces where subtle prompt adjustments can significantly influence the outcome.

Task Setup: We evaluate prompt optimization on two reasoning tasks: Object Counting from the Big Bench Hard benchmark (Suzgun et al., 2022; Srivastava et al., 2022) and grade-school math problem solving from the GSM8K dataset (Cobbe et al., 2021). For each task, when using the iterative optimization methods, we use a batch size of 3 across 12 optimization iterations, allowing the model to process a total of 36 training examples, randomly sampled with replacement. After each iteration, we validate the prompt using a validation set, and if the validation accuracy improves, we update the prompt accordingly. We compare the model's accuracy on the test set after all 12 iterations, using prompts generated by different optimization methods. Consistent with (Yuksekgonul et al., 2024), for both tasks, we use the **string-based exact match** metric, which looks at the final numerical value provided in the answer, and compares it to the ground truth answer. Detailed task setup is provided in Appendix B.

Baselines and LLM Backends: We evaluate HessianGrad against three key baselines:

- **Zero-shot Chain-of-Thought (CoT)** (Kojima et al., 2022; Wei et al., 2022): This baseline initializes all prompts using a zero-shot CoT strategy, where the model is prompted to "think step-by-step" before generating an answer. This approach is widely regarded as a strong baseline for reasoning tasks.
- **TextGrad** (Yuksekgonul et al., 2024): Textual feedback is treated as a first-order gradient to iteratively optimize prompts.
- **Momentum-Enhanced TextGrad (Yuksekgonul et al., 2024)**: This method extends the original TextGrad framework by incorporating momentum. This variant aims to overcome potential stagnation in the optimization process by enlarging updates to the prompt when previous feedbacks on the variable are similar.

Our experiments perform prompt optimization separately on four LLMs: gpt-3.5-turbo-0125, GPT-4, Gemini 1.5 Pro, and Llama 3.1 8B Instruct, with gpt-4o serving as the backend of the optimization system. This multi-model setup allows us to evaluate the effectiveness of the optimization methods across diverse architectures, ensuring a comprehensive assessment of their capabilities.

Results: As evidenced by Table 1, in both reasoning tasks, HessianGrad delivers a substantial improvement over the Zero-shot CoT prompt, underscoring its effectiveness across diverse datasets and model architectures. On the Object Counting task, with Llama 3.1 8B Instruct as the base model, HessianGrad outperforms TextGrad by achieving a 6% higher accuracy, demonstrating its superior ability to refine LLM responses. Similarly, on GSM8K, HessianGrad exceeds both TextGrad and M-Textgrad across most LLM backends, with an average performance increase of 2% over TextGrad. These results suggest that HessianGrad not only enhances the optimization process but also addresses the inherent limitations of first-order feedback in TextGrad, leading to more accurate and refined reasoning capabilities.

Universality: "HessianGrad's universality is evidenced by its consistent performance across all LLMs, including gpt-3.5-turbo-0125, GPT-4, and Llama 3.1 8B Instruct, where it delivers the highest accuracy with an average improvement of 5-7% compared to the baselines. However, there is one exception on the Gemini-1.5-Pro model, where HessianGrad slightly trails behind TextGrad. This small performance gap may be due to the use of GPT-4o to guide the Gemini-1.5-pro in the prompt optimization reasoning task. Given that Gemini-1.5-pro may exhibit more sophisticated reasoning

Table 1: Prompt optimization results for reasoning tasks for various LLMs, with gpt-4o as the optimization engine. The values in parentheses represent the relative improvement in accuracy of the method compared to TextGrad.

Dataset	Models	Accuracy % (Improv. over TextGrad)			
		CoT	TextGrad	M-TextGrad	HessianGrad
Object Counting	GPT-3.5	77.8 (15.3%↓)	91.9 (-)	92.1 (0.2%↑)	**95.5 (3.9%↑)**
	GPT-4	92.1 (2.2%↓)	94.2 (-)	90.0 (4.5%↓)	**96.3 (2.2%↑)**
	Gemini 1.5 Pro	94.0 (0.0%)	94.0 (-)	94.0 (0.0%)	94.0 (0.0%)
	Llama 3.1 8B Instruct	65.0 (15.6%↓)	77.0 (-)	80.0 (3.9%↑)	**83.0 (7.8%↑)**
GSM8k	GPT-3.5	72.9 (9.9%↓)	80.9 (-)	82.1 (1.5%↑)	**85.9 (6.2%↑)**
	GPT-4	92.6 (0.4%↓)	93.0 (-)	93.9 (1.0%↑)	**94.5 (1.6%↑)**
	Gemini 1.5 Pro	92.9 (0.4%↓)	93.3 (-)	**93.9 (0.6%↑)**	93.0 (0.3%↓)
	Llama 3.1 8B Instruct	84.6 (0.0%)	84.6 (-)	84.6 (0.0%)	84.6 (0.0%)

capabilities than GPT-4o in this specific scenario, the transfer of guidance from GPT-4o could have introduced suboptimal adjustments, leading to a slight degradation in performance. Despite this, HessianGrad remains highly adaptable and effective across diverse LLM backends, reaffirming its versatility as a powerful optimization tool.

We notice that on the GSM8K dataset with Llama-3.1, all methods show stagnant performance, likely due to the model's saturation on this task, leaving little room for further improvement from optimization methods. Despite this, HessianGrad demonstrates clear advantages in enhancing weaker, cost-effective models like gpt-3.5-turbo-0125, using feedback from stronger models such as gpt-4o. By incurring a one-time optimization cost, HessianGrad provides optimized prompts for weaker models, offering significant performance gains without the high inference costs of stronger models. This efficiency makes it an ideal solution for cost-sensitive AI deployment.

4.2 Solution optimization

We proceed to evaluate HessianGrad on the solution optimization task. This task aims to refine and improve the solution to complex scientific or technical problems, such as questions in quantum mechanics or organic chemistry. The solution will evolve dynamically through self-evaluation and critique, challenging the LLM to continually refine its responses. This process aligns with test-time training (Sun et al., 2020; 2024), where models are refined during testing, as well as with recent progress in self-refinement for reasoning tasks (Yao et al., 2022; Madaan et al., 2024; Shinn et al., 2024), which have demonstrated efficacy in iterative problem-solving.

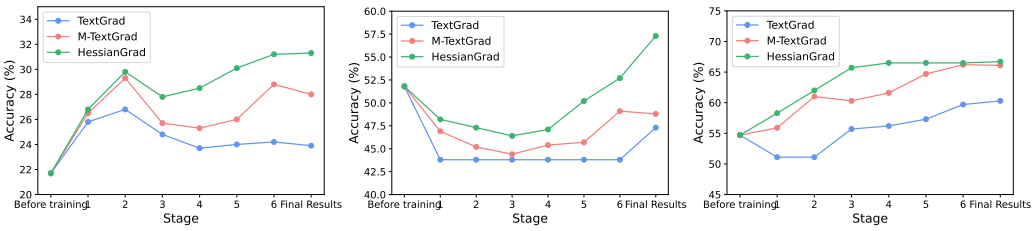
Task Setup: We evaluate solution optimization on two challenging benchmarks: Google-proof Question Answering (GPQA)(Rein et al., 2023), which consists of expert-level multiple-choice questions in physics, biology, and chemistry, and two subsets of the MMLU benchmark(Hendrycks et al., 2020), specifically focused on Machine Learning and College Physics. GPQA is a highly difficult benchmark, with experts achieving 81% accuracy and skilled non-experts reaching only 22%, highlighting the challenge of the questions. Performance of LLMs on these benchmarks has not yet saturated, making them ideal for benchmarking solution refinement. We use three iterations of optimization for each question when using the iterative optimization methods. The final answer is determined through majority voting across all iterations for all the iterative optimization methods. Consistent with (Yuksekgonul et al., 2024), we use the **string-based exact match** metric. Detailed task setup is in Appendix C.

Baselines and LLM Backends: We compare HessianGrad against three primary baselines for solution optimization: Chain-of-Thought (CoT) (Kojima et al., 2022; Wei et al., 2022), TextGrad (Yuksekgonul et al., 2024) and Momentum-Enhanced TextGrad. All methods use the Llama 3.1 8B Instruct model as the backend. Detailed baseline configurations and prompting exemplars can be found in Appendix C.

Results: As shown in Table 2, across all benchmarks, HessianGrad significantly improves the performance of Llama 3.1 8B Instruct compared to all baselines. On average, across the three benchmarks, HessianGrad achieves a 17.79% relative improvement in final accuracy over TextGrad. This substantial gain highlights the effectiveness of incorporating second-order gradients into the optimization process, enabling more precise adjustments and greater performance gains on solution optimization tasks.

Table 2: Solution optimization results for Llama 3.1 8B Instruct, with itself as the optimization engine. The values in parentheses represent the relative improvement of the method compared to TextGrad.

Dataset	Stage	Accuracy % (Improv. over TextGrad)			
		CoT	TextGrad	M-TextGrad	HessianGrad
Google-proof QA	Before Training	21.7 (0.0%)	21.7 (-)	21.7 (0.0%)	21.7 (0.0%)
	1st Iteration	-	25.8 (-)	26.5 (2.7%↑)	**26.8 (3.88%↑)**
	2nd Iteration	-	26.8 (-)	29.3 (9.3%↑)	**29.8 (11.19%↑)**
	3rd Iteration	-	24.8 (-)	25.7 (3.6%↑)	**27.8 (12.10%↑)**
	Final Results	21.7 (8.4%↓)	23.7 (-)	25.1 (5.9%↑)	**28.3 (19.41%↑)**
MMLU-Machine Learning	Before Training	51.8 (0.0%)	51.8 (-)	51.8 (0.0%)	51.8 (0.0%)
	1st Iteration	-	43.8 (-)	46.9 (7.1%↑)	**48.2 (10.05%↑)**
	2nd Iteration	-	43.8 (-)	45.2 (3.2%↑)	**47.3 (7.99%↑)**
	3rd Iteration	-	43.8 (-)	44.4 (1.4%↑)	**46.4 (5.94%↑)**
	Final Results	51.8 (9.5%↑)	47.3 (-)	47.4 (0.2%↑)	**57.1 (20.72%↑)**
MMLU-College Physics	Before Training	54.7 (0.0%)	54.7 (-)	54.7 (0.0%)	54.7 (0.0%)
	1st Iteration	-	51.1 (-)	55.9 (9.4%↑)	**58.3 (14.1%↑)**
	2nd Iteration	-	51.1 (-)	61.0 (19.4%↑)	**62.0 (21.3%↑)**
	3rd Iteration	-	55.7 (-)	60.3 (8.3%↑)	**65.7 (18.0%↑)**
	Final Results	54.7 (9.3%↓)	60.3 (-)	61.6 (2.2%↑)	**66.4 (10.1%↑)**



(a) *Google-proof QA* (b) *MMLU-Machine Learning* (c) *MMLU-College Physics*

Figure 2: Loss curves w.r.t. accuracy on solution optimization task.

Deterioration in TextGrad: Interestingly, we observe performance deterioration with TextGrad on the MMLU benchmark, where both intermediate and final results are worse than the initial state. This highlights a key limitation of first-order optimization: relying solely on immediate feedback without accounting for curvature can lead to unstable optimization, potentially causing the model's performance to degrade over time.

Fluctuations in Momentum-Based TextGrad: While Momentum-Based TextGrad avoids stagnation seen in TextGrad method, its performance fluctuates significantly across iterations. This is due to its reliance on larger, varied changes when feedback becomes repetitive, which can lead to overshooting and destabilization. Though it helps break feedback loops, momentum-based methods often amplify change without tracking the precise evolution of responses.

In contrast, HessianGrad overcomes these limitations by capturing gradient curvature, enabling better global adjustments and avoiding stagnation, proving its superiority in complex optimization scenarios. These results illustrate that by spending additional computational resources during test-time, HessianGrad significantly enhances performance, even for advanced models. Its iterative, second-order optimization approach makes it highly effective across diverse tasks, ensuring robust and versatile optimization for AI systems requiring high performance and accuracy.

Empirical Analysis of Loss Curves. To evaluate the effectiveness of HessianGrad's second-order-inspired behavior, we further analyze the optimization curves on solution optimization. Since explicit numerical loss values are unavailable, we use test accuracy as a proxy for loss to approximate the optimization dynamics. To better illustrate the optimization trends and provide clearer insights into the iterative process, we extend the number of iterations to 6 and plot the resulting loss curves in Figure 2. The results demonstrate the following key effects:

- **Escaping Local Optima:** HessianGrad consistently surpasses performance plateaus, as shown in Figure 2 (b), by leveraging cumulative response dynamics to guide updates.

- **Stabilizing Updates:** Unlike baselines such as M-TextGrad, which exhibit oscillations and instability (*e.g.,* Figure 2 (c)), HessianGrad achieves smoother optimization trajectories, demonstrating its robustness in maintaining stability.
- **Improved Performance in Complex Scenarios:** Proxy loss curves across all datasets highlight HessianGrad's ability to make meaningful adjustments over iterative refinements.

These results validate that HessianGrad effectively simulates and operationalizes second-order-inspired effects, enhancing optimization outcomes without relying on explicit numerical second-order computations.

4.3 Code Optimization

The Code Optimization task aims to refine code snippets to improve their correctness and runtime efficiency, often with limited supervision from local tests and iterative self-evaluation. This task is also well-suited for evaluating optimization techniques as it requires handling intricate problem constraints and optimizing through iterative adjustments.

Task Setup: We evaluate code optimization using the LeetCode Hard dataset (Shinn et al., 2024), an online platform featuring coding challenges commonly used for technical interview preparation. The primary metric for this task is the **Completion Rate**, which measures the percentage of problems for which all test cases are passed, calculated as $\frac{\text{Number of problems passed}}{\text{Total number of problems}}$. Since LeetCode test cases are not publicly available, generated code is submitted to the LeetCode platform for evaluation on these unseen test cases. Results are averaged over multiple runs for robustness. Additional details of the task setup are provided in Appendix D.

Baselines and LLM Backends: We evaluate HessianGrad against four key baselines on the LeetCode Hard dataset using Llama 3.1 8B Instruct model as the backend:

- **Zero-shot Baseline**: We run a zero-shot baseline, following the same zero-shot baseline setup described in (Shinn et al., 2024).
- **Reflexion** (Shinn et al., 2024): The state-of-the-art method for code optimization, which prompts an LLM to self-reflect on generated code snippets and errors based on candidate unit tests. Reflexion then prompts the LLM to update the code based on this self-reflection. We run Reflexion using a one-shot setting, with one in-context demonstration to guide its behavior.
- **TextGrad** and **Momentum-Enhanced TextGrad**: We run TextGrad, M-TextGrad, and HessianGrad in a zero-shot setting without demonstrations, refining the code based solely on feedback from each iteration.

Table 3: Code optimization results (averaged over 5 seeds) for Llama 3.1 8B Instruct, with itself as the optimization engine. The values in parentheses represent the relative improvement in completion rate of the method compared to TextGrad.

Dataset	Method	Completion Rate (Improv. over TextGrad)
LeetCode Hard	Zero-shot	0.12 (50%↓)
	Reflexion (1 demonstration, 5 iterations)	0.20 ± 0.002 (16.67%↑)
	TextGrad (0 demonstrations, 5 iterations)	0.24 ± 0.005 (-)
	M-TextGrad (0 demonstrations, 5 iterations)	0.25 ± 0.003 (4.17%↑)
	HessianGrad (0 demonstrations, 5 iterations)	**0.31 ± 0.006 (29.17%↑)**

Results: As presented in Table 3, HessianGrad demonstrates the strongest performance on the LeetCode Hard dataset, achieving a completion rate of 31%, which represents a 29.17% improvement over the baseline TextGrad. This significantly surpasses Reflexion's performance, which showed a 16.67% improvement, and Momentum-Enhanced TextGrad, which only offered a marginal 4.17% increase over TextGrad. These results highlight the effectiveness and robustness of HessianGrad in refining code snippets, especially in challenging coding problems where more nuanced optimization techniques are required. While Momentum-Enhanced TextGrad does provide some improvement, its performance lags considerably behind HessianGrad.

4.4 Ablation Study

Given the simplicity of our method, there are no complex components that can be eliminated for a traditional ablation. Instead, we conduct an ablation study by testing different prompt designs to evaluate their impact on performance. Specifically, we compare our HessianGrad prompt, a variant

Table 5: Comparison of computational resources (GPU memory and runtime) for HessianGrad and baseline methods across tasks.

Dataset	Stage	Time per Iteration (s)			Total Time to Converge (s)			GPU Usage (GB)		
		TextGrad	M-TextGrad	HessianGrad	TextGrad	M-TextGrad	HessianGrad	TextGrad	M-TextGrad	HessianGrad
Prompt Optimization	Objective Counting	92.144	110.721	137.815	276.450	110.732	137.821	3.23	3.24	3.23
	GSM8K	135.184	152.423	176.538	1351.85	1219.393	1235.774	3.23	3.23	3.23
Solution Optimization	Google-proof QA	153.522	178.879	197.235	614.216	1091.162	453.641	3.24	3.23	3.24
	MMLU-Machine Learning	172.429	207.819	223.807	896.631	685.803	626.659	3.24	3.24	3.24
	MMLU-College Physics	188.116	225.631	245.167	1636.612	1308.662	1054.229	3.24	3.24	3.24
Code Optimization	LeetCode Hard	1078.783	1241.917	1352.174	18986.655	18472.411	15820.489	6.46	6.46	6.46

of this prompt, and the prompt used in TextGrad to highlight the effectiveness of our approach. More details on the prompts can be found in Appendix E.

- **HessianGrad Prompt:** This is the prompt carefully designed for HessianGrad, which takes into account both immediate feedback and the evolution of responses across iterations.
- **Variant Prompt:** It differs from our method by directly instructing the LLM to generate more diverse responses, effectively pushing it towards greater variation with each iteration.
- **TextGrad Prompt:** Serving as the baseline, TextGrad's prompt focuses primarily on immediate feedback, making adjustments based solely on the latest response.

We conducted experiments on objective counting in the prompt optimization task, with results shown in Table 6.

Table 4: Prompt optimization results for reasoning tasks for various LLMs, with gpt-4o as the optimization engine.

Dataset	Method	**Accuracy %**
Object Counting	TextGrad	77.0
	Variant	80.0
	HessianGrad	**83.0**

On the Object Counting task, the Variant prompt surpasses TextGrad by encouraging larger, more diverse shifts in the response space, enabling the model to explore more distinct outputs with each iteration. HessianGrad, on the other hand, achieves even better results by promoting stable, iterative refinement rather than abrupt changes. While the Variant's strategy can lead to sudden, exaggerated shifts, HessianGrad ensures smoother, controlled optimization, gradually fine-tuning responses for greater accuracy.

4.5 Comparison of Computational Resources

To analyze the computational efficiency of HessianGrad, we compare its GPU memory usage and runtime against baseline methods across three task categories. All experiments use Llama-3.1-8B as the base LLM, running on a setup with 4 NVIDIA 3090 GPUs. The results are shown in Table 5.

We observe that while HessianGrad involves slightly higher per-iteration runtime due to its second-order optimization-inspired design, it converges in fewer iterations, resulting in significant overall savings. The detailed findings are as follows:

- On Object Counting dataset, HessianGrad reduces total runtime by 50% compared to TextGrad by converging in fewer iterations despite slightly higher per-iteration costs.
- For solution optimization task, HessianGrad achieves 26.14% lower total runtime than TextGrad, while M-TextGrad incurs 77.65% higher runtime due to instability.
- For code optimization task, HessianGrad reduces total runtime by 16.67% compared to baselines.
- For GPU memory usage, HessianGrad demonstrates similar requirements to baseline methods, indicating no significant increase in computational resources.

5 Conclusion

In this paper, we introduced HessianGrad, an optimization framework that extends traditional methods by considering the evolution of responses over multiple iterations. Instead of focusing only on immediate feedback, HessianGrad incorporates insights from the similarity between consecutive responses, akin to how second-order information is used in classic optimization. This approach allows for more informed adjustments, addressing issues like stagnation and instability seen in earlier methods. By capturing these iterative changes, HessianGrad achieves more stable and consistent improvements across tasks, particularly in complex scenarios where simpler methods fall short. This makes HessianGrad a valuable step forward in effective optimization for AI systems.

REFERENCES

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

Anthropic. Introducing claude. 2023. URL `https://www.anthropic.com/index/introducing-claude`.

Tom B Brown. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020.

Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, et al. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Qingyan Guo, Rui Wang, Junliang Guo, Bei Li, Kaitao Song, Xu Tan, Guoqing Liu, Jiang Bian, and Yujiu Yang. Connecting large language models with evolutionary algorithms yields powerful prompt optimizers. *arXiv preprint arXiv:2309.08532*, 2023.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. *arXiv preprint arXiv:2009.03300*, 2020.

Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *science*, 313(5786):504–507, 2006.

Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Saiful Haq, Ashutosh Sharma, Thomas T Joshi, Hanna Moazam, Heather Miller, et al. Dspy: Compiling declarative language model calls into state-of-the-art pipelines. In *The Twelfth International Conference on Learning Representations*, 2024.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35: 22199–22213, 2022.

Qing Lyu, Shreya Havaldar, Adam Stein, Li Zhang, Delip Rao, Eric Wong, Marianna Apidianaki, and Chris Callison-Burch. Faithful chain-of-thought reasoning. *arXiv preprint arXiv:2301.13379*, 2023.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. Self-refine: Iterative refinement with self-feedback. *Advances in Neural Information Processing Systems*, 36, 2024.

Archiki Prasad, Peter Hase, Xiang Zhou, and Mohit Bansal. Grips: Gradient-free, edit-based instruction search for prompting large language models. *arXiv preprint arXiv:2203.07281*, 2022.

Reid Pryzant, Richard Diehl Martinez, Nathan Dass, Sadao Kurohashi, Dan Jurafsky, and Diyi Yang. Automatically neutralizing subjective bias in text. In *Proceedings of the aaai conference on artificial intelligence*, volume 34, pp. 480–489, 2020.

Reid Pryzant, Dan Iter, Jerry Li, Yin Tat Lee, Chenguang Zhu, and Michael Zeng. Automatic prompt optimization with" gradient descent" and beam search. *arXiv preprint arXiv:2305.03495*, 2023.

David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R Bowman. Gpqa: A graduate-level google-proof q&a benchmark. *arXiv preprint arXiv:2311.12022*, 2023.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. Reflexion: Language agents with verbal reinforcement learning. *Advances in Neural Information Processing Systems*, 36, 2024.

Aarohi Srivastava, Abhinav Rastogi, Abhishek Rao, Abu Awal Md Shoeb, Abubakar Abid, Adam Fisch, Adam R Brown, Adam Santoro, Aditya Gupta, Adrià Garriga-Alonso, et al. Beyond the imitation game: Quantifying and extrapolating the capabilities of language models. *arXiv preprint arXiv:2206.04615*, 2022.

Yu Sun, Xiaolong Wang, Zhuang Liu, John Miller, Alexei Efros, and Moritz Hardt. Test-time training with self-supervision for generalization under distribution shifts. In *International conference on machine learning*, pp. 9229–9248. PMLR, 2020.

Yu Sun, Xinhao Li, Karan Dalal, Jiarui Xu, Arjun Vikram, Genghan Zhang, Yann Dubois, Xinlei Chen, Xiaolong Wang, Sanmi Koyejo, et al. Learning to (learn at test time): Rnns with expressive hidden states. *arXiv preprint arXiv:2407.04620*, 2024.

Mirac Suzgun, Nathan Scales, Nathanael Schärli, Sebastian Gehrmann, Yi Tay, Hyung Won Chung, Aakanksha Chowdhery, Quoc V Le, Ed H Chi, Denny Zhou, et al. Challenging big-bench tasks and whether chain-of-thought can solve them. *arXiv preprint arXiv:2210.09261*, 2022.

Gemini Team, Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, et al. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=Bb4VGOWELI.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. React: Synergizing reasoning and acting in language models. *arXiv preprint arXiv:2210.03629*, 2022.

Mert Yuksekgonul, Federico Bianchi, Joseph Boen, Sheng Liu, Zhi Huang, Carlos Guestrin, and James Zou. Textgrad: Automatic" differentiation" via text. *arXiv preprint arXiv:2406.07496*, 2024.

Shaokun Zhang, Jieyu Zhang, Jiale Liu, Linxin Song, Chi Wang, Ranjay Krishna, and Qingyun Wu. Offline training of language model agents with functions as learnable weights. In *Forty-first International Conference on Machine Learning*.

Wenqi Zhang, Ke Tang, Hai Wu, Mengna Wang, Yongliang Shen, Guiyang Hou, Zeqi Tan, Peng Li, Yueting Zhuang, and Weiming Lu. Agent-pro: Learning to evolve via policy-level reflection and optimization. *arXiv preprint arXiv:2402.17574*, 2024.

Lianmin Zheng, Wei-Lin Chiang, Ying Sheng, Siyuan Zhuang, Zhanghao Wu, Yonghao Zhuang, Zi Lin, Zhuohan Li, Dacheng Li, Eric Xing, et al. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in Neural Information Processing Systems*, 36, 2024.

Wangchunshu Zhou, Yixin Ou, Shengwei Ding, Long Li, Jialong Wu, Tiannan Wang, Jiamin Chen, Shuai Wang, Xiaohua Xu, Ningyu Zhang, et al. Symbolic learning enables self-evolving agents. *arXiv preprint arXiv:2406.18532*, 2024.

Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and Jurgen Schmidhuber. Language agents as optimizable graphs. *arXiv preprint arXiv:2402.16823*, 2024.

A System Prompt Details for HessianGrad

In HessianGrad, we use system prompts designed to guide iterative response refinement. The prompts focus on comparing the current response with previous iterations, emphasizing gradual, thoughtful evolution. They request the model to provide feedback not only on immediate changes but also on patterns observed across multiple iterations.

we use the following glossary to the system prompt:

GLOSSARY TEXT

Glossary of tags that will be sent to you:
- <LM_SYSTEM_PROMPT>: The system prompt for the language model.
- <LM_INPUT>: The input to the language model.
- <LM_OUTPUT>: The output of the language model.
- <FEEDBACK>: The feedback to the variable.
- <CONVERSATION>: The conversation history.
- <FOCUS>: The focus of the optimization.
- <ROLE>: The role description of the variable.

The Optimize Prompts details are as follows:

OPTIMIZER SYSTEM PROMPT

"You are part of an optimization system that improves text (i.e., variable) by analyzing how the responses evolve across multiple iterations. "
"Your goal is not just to make a single improvement, but to ensure that the variable evolves naturally and meaningfully over time. "
"Focus on adjusting the variable in a way that each step introduces thoughtful, measured changes based on past iterations, rather than drastic shifts. "
"The feedback provided will help guide these adjustments, but ensure that your improvements maintain coherence and contextual alignment. "
"You MUST give your response by sending the improved variable between {new_variable_start_tag} {{improved variable}} {new_variable_end_tag} tags. "
f"{GLOSSARY_TEXT}"

Textual Gradient Descent Prompt Prefix

"Here is the role of the variable you will improve: <ROLE>{variable_desc}</ROLE>."
"The variable is the text within the following span: <VARIABLE> {variable_short} </VARIABLE>" "Here is the context and feedback we received for the variable:"
"<CONTEXT>{variable_grad}</CONTEXT>"
"Additionally, reflect on how the responses to this variable have evolved across iterations:"
"<PAST_ITERATIONS>{past_values}</PAST_ITERATIONS>"
"Make nuanced improvements, keeping in mind that too-similar responses suggest insufficient change, but avoid making overly large changes. "
"Ensure that the response evolves in a coherent and thoughtful manner that aligns with the context, feedback, and past responses."

The following is how we save gradients to the variable.

GRADIENT TEMPLATE

"Here is a conversation:<CONVERSATION>{context}</CONVERSATION>"
"This conversation is part of a larger system. The output is used as {response_desc}. "
"Here is the feedback we received for {variable_desc} in the conversation:<FEEDBACK>{feedback}</FEEDBACK>"
"Additionally, consider how the responses to this variable have changed across previous iterations:"
"<PAST_ITERATIONS>{past_values}</PAST_ITERATIONS>"
"Make sure future responses reflect a meaningful, gradual evolution based on these past iterations, encouraging thoughtful progress rather than drastic shifts."

B PROMPT OPTIMIZATION

For the dataset split, we follow the settings used in TextGrad (Yuksekgonul et al., 2024). The Big Bench Hard Object Counting dataset is divided into 50/100/100 samples for train/validation/test, respectively. For GSM8K, we adopt the split from DSPy (Khattab et al., 2024), using 200/300/1399 samples for train/validation/test. In each task, we limit the training set to 36 samples, consistent with the TextGrad setup. Example queries for each dataset are shown below:

Example Query for Big Bench Hard Object Counting

I have an apple, three bananas, a strawberry, a peach, three oranges, a plum, a raspberry, two grapes, a nectarine, and a blackberry. How many fruits do I have?

Example Query for GSM8K

Natalia sold clips to 48 of her friends in April, and then she sold half as many clips in May. How many clips did Natalia sell altogether in April and May?

For the momentum-enhanced TextGrad baseline, to ensure a fair comparison with HessianGrad, which accounts for responses from all previous iterations, we set the momentum window to 12 so that momentum-enhanced TextGrad has access to gradients from all prior iterations.

Regarding specific hyperparameters for the LLMs, we set the temperature to 0 (1×10^{-6} for Llama 3.1 8B Instruct), allow a maximum of 2000 new tokens, and use a top-p value of 0.99.

C SOLUTION OPTIMIZATION

For the solution optimization task, we follow the experimental setup outlined by TextGrad (Yuksekgonul et al., 2024). This ensures fair comparisons across all experiments. We evaluate on two benchmarks: Google-Proof Question Answering (GPQA) (Rein et al., 2023) and two subsets from the MMLU benchmark (Hendrycks et al., 2020), Machine Learning and College Physics. Following the simple-evals repository practice, we employ string matching to extract the final answer (one of ABCD) and compare it to the ground truth. The datasets comprise 198 questions in the GPQA Diamond subset, 112 in MMLU Machine Learning, and 92 in MMLU College Physics. We compare HessianGrad against three primary baselines for solution optimization:

- Chain-of-Thought (CoT) (Kojima et al., 2022; Wei et al., 2022): This baseline serves as our initial baseline. This method employs a step-by-step reasoning process, providing a strong foundation for comparison in complex problem-solving tasks.
- TextGrad (Yuksekgonul et al., 2024): This method leverages textual gradients to iteratively refine solutions. For the solution optimization task, we apply three iterations of test-time updates using TextGrad, refining the solution at each step. The process involves making one call to GPT-4o to evaluate the test-time loss, another call to collect gradients, and a final call to update the solution accordingly.

- Momentum-Enhanced Textgrad: This model builds upon the original TextGrad framework by incorporating momentum. This variant aims to overcome potential stagnation in the optimization process by adjusting the magnitude of updates based on the history of previous iterations. Like TextGrad, we apply three iterations of test-time updates for this method.

For TextGrad, Momentum-Enhanced TextGrad, and our proposed HessianGrad method, we determine the final answer through majority voting across all iterations. This approach ensures that we capture the best performance from each method over multiple refinement steps.

Example Query for GPQA Diamond

Answer the following multiple choice question. The last line of your response should be of the following format: 'Answer: $LETTER' (without quotes) where LETTER is one of ABCD. Think step by step before answering.
A) A = cyclohexane-1,3,5-trione, B = dimethyl fumarate
B) A = benzoquinone, B = dimethyl fumarate
C) A = benzoquinone, B = methyl 2-hydroxypropanoate
D) A = cyclohexane-1,3,5-trione, B = methyl 2-hydroxypropanoate

Example Comparison of TextGrad and HessianGrad

Answer the following multiple-choice question... Select the suitable reagents for the following mentioned reactions. butan-2-one + NaCN + A —> 2-hydroxy-2-methylbutanenitrile 2-(4-benzylphenyl)-2-hydroxybutanenitrile + B (H2O) —> 2-(4-benzylphenyl)-2-hydroxybutanoic acid.
A) A = NaHSO3, B = CH3COOH
B) A = H3O+, B = HCl
C) A = NaHSO3, B = HCl
D) A = H3O+, B = CH3COOH
(Ground Truth Answer: C)
TextGrad Response: ...let's evaluate the options: ... Option B: A = H3O+, B = HCl - H3O+ (hydronium ion) is a suitable proton source for cyanohydrin formation. HCl is a strong acid and commonly used for hydrolyzing nitriles to carboxylic acids. Option C: A = NaHSO3, B = HCl **NaHSO3 is not typically used in cyanohydrin formation.** HCl is suitable for hydrolyzing nitriles. ... Based on the analysis, the most suitable reagents for the reactions are: A = H3O+ for the cyanohydrin formation. B = HCl for the hydrolysis of the nitrile to a carboxylic acid. Therefore, the correct answer is: Answer: B.
HessianGrad Response: ...The first reaction is the addition of cyanide ion (CN-) to butan-2-one to form 2-hydroxy-2-methylbutanenitrile. This is an example of a cyanohydrin reaction, which typically involves the addition of cyanide ion to an aldehyde or ketone in the presence of a catalyst such as sodium bisulfite (NaHSO3).The second reaction is the hydrolysis of 2-(4-benzylphenyl)-2-hydroxybutanenitrile to form 2-(4-benzylphenyl)-2-hydroxybutanoic acid. This is an example of a hydrolysis reaction, which typically involves the addition of water (H2O) in the presence of a strong acid such as hydrochloric acid (HCl). **Based on this analysis, the suitable reagents for the reactions are:A = NaHSO3 (sodium bisulfite) for the first reaction. B = HCl (hydrochloric acid) for the second reaction.** So, the correct answer is: Answer: C.

Case Study: In this example, TextGrad incorrectly selects H3O+ for cyanohydrin formation, misunderstanding the role of NaHSO3, which is a typical catalyst in such reactions. Its reliance on first-order feedback leads to an erroneous conclusion. In contrast, HessianGrad correctly identifies NaHSO3 as the catalyst for the first reaction and HCl for the hydrolysis in the second reaction. By leveraging second-order gradients, HessianGrad better captures the complexities of the chemical mechanisms, leading to the correct answer.

D CODE OPTIMIZATION

In the code optimization task, we primarily rely on the settings from previous work, particularly TextGradYuksekgonul et al. (2024), to ensure a fair comparison across experiments. Specifically, we adopt the version of ReflexionShinn et al. (2024) as used in TextGrad, which includes minor modifications for compatibility within the TextGrad framework.

For the baselines, we employ two key approaches:

- Reflexion Shinn et al. (2024): In this setup, the language model is guided by a one-shot prompt instructing it to provide feedback on the code it generates. The process begins by generating an initial solution based on a provided code prompt. The solution is first tested locally, and if it passes, it is then submitted to the LeetCode platform for a more rigorous evaluation using harder test cases. If the local tests fail, Reflexion is used to request feedback from the model to refine the code. This feedback-guided optimization is repeated for up to 5 iterations, during which the model continuously improves the code until it successfully passes local tests and is ready for submission.
- TextGrad Yuksekgonul et al. (2024): This baseline runs 5 independent trials, each with five seeds with [15, 17, 21, 55, 91], and averages the results to ensure consistency. During each optimization iteration, TextGrad performs three key operations: first, it makes a call to GPT-4o to evaluate the test time loss; next, it collects gradients based on the loss; finally, the code snippet is updated according to the gradients. This process repeats, optimizing the code over several iterations to minimize the test time loss and improve performance on the test cases.

The number of coding problems in LeetCodeHard is 39. We set By following the setup from TextGrad, we ensure that both Reflexion and TextGrad are evaluated under the same conditions, facilitating a fair and consistent comparison with these two baselines and HessianGrad.

Example Query for LeetCode Hard

```
def minimumTime(grid: List[List[int]]) -> int:
"""
You are given a ‘m x n’ matrix ‘grid’ consisting of non-negative integers where ‘grid[row][col]‘
represents the minimum time required to be able to visit the cell ‘(row, col)’, which means
you can visit the cell ‘(row, col)’ only when the time you visit it is greater than or equal to
‘grid[row][col]’.
You are standing in the top-left cell of the matrix in the ‘0th’ second, and you must move to
any adjacent cell in the four directions: up, down, left, and right. Each move you make takes
1 second. Return the minimum time required in which you can visit the bottom-right cell of
the matrix. If you cannot visit the bottom-right cell, then return ‘-1’.
Example 1:
Input: grid = [[0,1,3,2],[5,1,2,5],[4,3,8,6]]
Output: 7
Explanation:
One of the paths that we can take is the following:
- at t = 0, we are on the cell (0,0).
- at t = 1, we move to the cell (0,1). It is possible because grid[0][1] <= 1.
- at t = 2, we move to the cell (1,1). It is possible because grid[1][1] <= 2.
- at t = 3, we move to the cell (1,2). It is possible because grid[1][2] <= 3.
- at t = 4, we move to the cell (1,1). It is possible because grid[1][1] <= 4.
- at t = 5, we move to the cell (1,2). It is possible because grid[1][2] <= 5.
- at t = 6, we move to the cell (1,3). It is possible because grid[1][3] <= 6.
- at t = 7, we move to the cell (2,3). It is possible because grid[2][3] <= 7.
The final time is 7. It can be shown that it is the minimum time possible.
Example 2:
Input: grid = [[0,2,4],[3,2,1],[1,0,4]]
Output: -1
Explanation:
There is no path from the top left to the bottom-right cell. Constraints:
* ‘m == grid.length’
```

* ‘n == grid[i].length’
* ‘2 <= m, n <= 1000’
* ‘4 <= m * n <= 105’
* ‘0 <= grid[i][j] <= 105’
* ‘grid[0][0] == 0’
"""

Example Comparison of TextGrad and HessianGrad

Task Description:
"""
You have k bags. You are given a **0-indexed** integer array $weights$ where $weights[i]$ is the weight of the i^{th} marble. You are also given the integer k.
Divide the marbles into the k bags according to the following rules:
- No bag is empty.
- If the i^{th} marble and j^{th} marble are in a bag, then all marbles with an index between the i^{th} and j^{th} indices should also be in that same bag.
- If a bag consists of all the marbles with an index from i to j inclusively, then the cost of the bag is $weights[i] + weights[j]$.

The score after distributing the marbles is the sum of the costs of all the k bags.
Return the **difference** between the **maximum** and **minimum** scores among marble distributions.
Example 1:
Input: weights = [1,3,5,1], k = 2
Output: 4
Explanation:
The distribution [1],[3,5,1] results in the minimal score of (1+1) + (3+1) = 6.
The distribution [1,3],[5,1], results in the maximal score of (1+3) + (5+1) = 10.
Thus, we return their difference 10 - 6 = 4.
Example 2:
Input: weights = [1, 3], k = 2
Output: 0
Explanation:
The only distribution possible is [1],[3].
Since both the maximal and minimal score are the same, we return 0.
"""
def putMarbles(weights: List[int], k: int) -> int:
Code Generated by TextGrad:

```
from typing import List

def putMarbles(weights: List[int], k: int) -> int:
    n = len(weights)
    if k == 1:
        return 0

    # Calculate the cost of each possible bag
    costs = [weights[i] + weights[i + 1] for i in range(n - 1)]

    # Sort the costs to find the minimum and maximum possible scores
    costs.sort()

    # The minimum score is the sum of the smallest k-1 costs
    min_score = sum(costs[:k-1])

    # The maximum score is the sum of the largest k-1 costs
    max_score = sum(costs[-(k-1):])

    return max_score - min_score
```

Tests that the code passed:

```
No tests passed.

**Tests that the code failed:**

assert putMarbles([1,3,5,1], 2) == 4 # ERROR: This unit test fails.
    Output was 2, but expected value was: 4
assert putMarbles([1,3], 2) == 0 # ERROR: This unit test fails.
    Output was -1, but expected value was: 0
```

Code Generated by HessianGrad

```python
from typing import List

def putMarbles(weights: List[int], k: int) -> int:
    n = len(weights)
    if k == 1 or k == n:
        return 0

    # Calculate the cost of each possible bag
    costs = [weights[i] + weights[i + 1] for i in range(n - 1)]

    # Sort the costs to find the minimum and maximum possible scores
    costs.sort()

    # The minimum score is the sum of the smallest k-1 costs
    min_score = sum(costs[:k-1])

    # The maximum score is the sum of the largest k-1 costs
    max_score = sum(costs[-(k-1):])

    return max_score - min_score
```

Tests that the code passed:

```
assert putMarbles([1,3,5,1], 2) == 4
assert putMarbles([1,3], 2) == 0

**Tests that the code failed:**

No tests failed.
```

E Abaltion Study

E.1 Ablation Study

Given the simplicity of our method, there are no complex components that can be eliminated for a traditional ablation. Instead, we conduct an ablation study by testing different prompt designs to evaluate their impact on performance. Specifically, using Llama-3.1-8B-Instruct as the LLM backend, we compare our HessianGrad prompt, a variant of this prompt, and the prompt used in TextGrad to highlight the effectiveness of our approach.

- **HessianGrad Prompt:** This is the prompt carefully designed for HessianGrad, which takes into account both immediate feedback and the evolution of responses across iterations.
- **Variant Prompt:** It differs from our method by directly instructing the LLM to generate more diverse responses, effectively pushing it towards greater variation with each iteration.
- **TextGrad Prompt:** Serving as the baseline, TextGrad's prompt focuses primarily on immediate feedback, making adjustments based solely on the latest response.

The detailed prompts for the variant are as follows:

OPTIMIZER SYSTEM PROMPT

"You are part of an optimization system that improves text (i.e., variable). "
"You will be asked to creatively and critically improve prompts, solutions to problems, code, or any other text-based variable. "
"You will receive some feedback, and use the feedback to improve the variable. "
"Pay attention to the role description of the variable, and the context in which it is used. "
"Importantly, focus on creating responses that are varied and diverse in nature. "
"You MUST give your response by sending the improved variable between {new_variable_start_tag} {{improved variable}} new_variable_end_tag tags. "
"The text you send between the tags will directly replace the variable."
f"{GLOSSARY_TEXT}"

Textual Gradient Descent Prompt Prefix

"Here is the role of the variable you will improve: <ROLE>{variable_desc}</ROLE>."
"The variable is the text within the following span: <VARIABLE> {variable_short} </VARIABLE>"
"Here is the context and feedback we got for the variable:"
"<CONTEXT>{variable_grad}</CONTEXT>"
"Improve the variable ({variable_desc}) using the feedback provided in <FEEDBACK> tags. "
"Ensure that your response introduces new and diverse ways of solving the problem or addressing the prompt."

The following is how we save gradients to the variable.

GRADIENT TEMPLATE

"Here is a conversation:<CONVERSATION>{context}</CONVERSATION>"
"This conversation is part of a larger system, where varied and creative outputs are important. "
"The output is used as {response_desc}. Here is the feedback we received for {variable_desc} in the conversation:"
"<FEEDBACK>{feedback}</FEEDBACK>"
"Encourage diversity in your improvements."

We conducted experiments on objective counting in the prompt optimization task, with results shown in Table 6.

Table 6: Prompt optimization results for reasoning tasks for various LLMs, with gpt-4o as the optimization engine.

Dataset	Method	**Accuracy %**
Object Counting	TextGrad	77.0
	Variant	80.0
	HessianGrad	**83.0**

On the Object Counting task, the Variant prompt surpasses TextGrad by encouraging larger, more diverse shifts in the response space, enabling the model to explore more distinct outputs with each iteration. HessianGrad, on the other hand, achieves even better results by promoting stable, iterative refinement rather than abrupt changes. While the Variant's strategy can lead to sudden, exaggerated shifts, HessianGrad ensures smoother, controlled optimization, gradually fine-tuning responses for greater accuracy.