Importance of Directional Feedback for LLM-based Optimizers

Anonymous Author(s)
Affiliation
Address
email

Abstract

We study the potential of using large language models (LLMs) as an interactive optimizer for solving maximization problems on a text space using natural language and numerical feedback. Inspired by the classical optimization literature, we classify the natural language feedback into directional and non-directional, where the former is a generalization of the first-order feedback to the natural language space. We find that LLMs are especially capable of optimization when they are provided with directional feedback. Based on this insight, we design a new LLM-based optimizer that synthesizes directional feedback from the historical optimization trace to achieve reliable improvement over iterations. Empirically, we show our LLM-based optimizer is more stable and efficient in solving optimization problems, from maximizing mathematical functions to optimizing prompts for writing poems, compared with existing techniques.

1 Introduction

Owing to their capability to produce a diverse range of outputs similar to those of humans, large language models (LLMs) are a powerful component for solving many difficult problems involving natural language, including planning [3], interacting with users, understanding documents [4], and producing executable code [2]. In addition to harnessing LLMs in these *generative* roles, several recent works have used LLMs for *optimization*. So far, these efforts, such as APO [7] and OPRO [11], have focused on optimization of a very specific kind – employing LLMs to produce prompts that improve (another) LLM's performance. In this work, we argue that LLMs' potential extends much further, to general optimization problems. We showcase that LLMs are capable of optimizing entities as dissimilar as mathematical functions and poems if they are provided with *directional feedback*.

The notions of directional and non-directional feedback arise naturally in many interactive decision-making domains and are tied to the classical optimization literature [1]. Typically, a numerical optimizer iterates over two steps. The first step aims to identify a "search direction" for improvement. This information is provided to the optimizer by an oracle, oftentimes a first-order oracle, and can be viewed as directional feedback. The second step decides what to change about the input. The applicability of various optimization methods depends on whether the directional feedback information is available or not. Scenarios without directional feedback are confined to black-box optimization methods such as evolutionary search [5], Bayesian optimization [6], or policy gradient [9]. However, when the directional feedback is available, one can choose the much more efficient gradient-based optimization method, such as stochastic gradient descent or exact line search [1]. This insight motivates our use of directional feedback in the realm of LLM-driven optimization.

As we show, the presence or absence of directional feedback and the possibility to access them is crucial for LLM-based optimization. For a systematic study of factors that make an optimization

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

process challenging, we choose one of the difficult tasks proposed in the work on OPRO (listed as failure cases in Appendix A of OPRO [11]) – navigating a bumpy loss function landscape. We discover that an LLM-based optimizer agent's performance varies with the information the feedback carries, and, given proper feedback, LLMs can strategically improve over past outputs, which makes this previously unsolvable task solvable. In addition, we demonstrate that using LLMs to "synthesize" feedback from a history of observations and prompts can help optimization too.

We also explore LLMs' optimization potential in a completely different setting. Given the importance of feedback type in the LLM-based optimization process and the lack of benchmarks that generate verbal feedback automatically, we create a synthetic poem writing environment, where one can programmatically create feedback for the LLMs. The poem environment is a family of tasks where an LLM is asked to write a poem. A distinguishing feature of this benchmark is that the poems must satisfy some constraints, such as the number of syllables per line. By leveraging and synthesizing feedback, we show that an LLM can sequentially optimize a poem-generation prompt to yield a high success rate of producing constraint-satisfying poems. Our results highlight the importance of understanding and studying the role of feedback in the broader LLM-based text optimization landscape.

2 Preliminaries: Prompt Optimization for LLM-based Agent

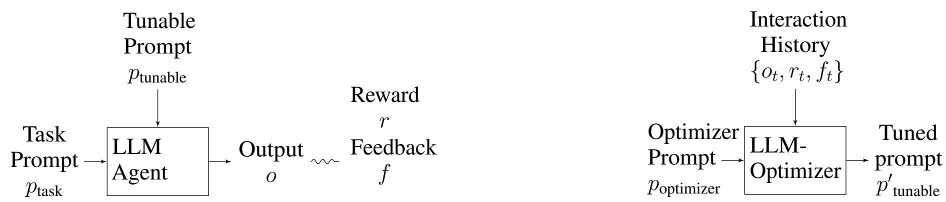
An LLM-based agent's behavior is modulated through the prompts used as inputs to the LLM. We describe the interactive decision-making problem encountered by an LLM-based agent, and how prompt optimization through interactive feedback can improve the agent over time. In the following, uppercase letters, e.g. X, denote random variables or sets. Lowercase letters denote realizations of the random variables or set elements, e.g. "$X = x$" states that a r.v. X takes on value x. Greek letters, e.g., ξ, denote parameters indexing probability distributions.

Consider an agent encountering a complex task such as generating a poem with logical constraints. The task is communicated to the agent via a text prompt $p_{\text{task}} =$ "Generate a poem with a rhyming scheme of abcba". The LLM produces output text $o_1 \sim \Pr_\tau(O \mid p_{\text{task}}, p_{\text{tunable}})$ (e.g., a poem, or plans, or executable code, or other texts as prompted), where τ captures LLM hyper-parameters like sampling temperature, and p_{tunable} contains any orchestrated text inputs from other modules surrounding the LLM. In the sequel, we will develop modules that will incorporate information gathered over time to update p_{tunable}. By analogy with tunable parameters of an ML model, we view p_{tunable} as the tunable "parameters" of an LLM-based agent.

Based on the generated output o_1, a scalar reward r_1 and optionally feedback f_1 are generated from the environment (e.g., human user response, or logs generated by executing code in a programming environment) and passed to the agent. For ease of notation, we assume $r \sim R$ and $f \sim F$, but we do not make specific assumptions of the underlying distributions. The reward can be a task success or failure boolean from the environment, or user-provided thumbs-up/down signal. This interaction process iterates $o_1 \rightsquigarrow \{r_1, f_1\}, \ldots, o_t \rightsquigarrow \{r_t, f_t\}$, until the environment terminates the interaction session. Figure 1a illustrates the interaction process; for example, in Minecraft Voyager [10] a prompt $p_{\text{task}} =$ "Build a house" is translated into a code-generation request using internal orchestration that prepends a specific p_{tunable}[1]. The produced code o_t is executed in the Minecraft environment to generate error/debug/return messages f_t as well as task completion flag r_t that are returned to Voyager to refine the code o_{t+1} in subsequent iterations. The interaction session ends when the user prompting the Voyager agent terminates it. Note that p_{task} can be interactively updated within a session (e.g., user providing additional hints or rephrasing the task), and we only assume that the rewards and feedbacks observed are consistent with the task that the agent is prompted to solve.

The LLM-Optimizer is a specific instance of an LLM-based agent. It can be used to improve another LLM-based agent using collected experience so that the generated outputs have higher expected reward $\mathbb{E}_o\left[r \mid o, p_{\text{task}}\right]$. The LLM-Optimizer takes a collection of Output-Reward-Feedback (o, r, f) tuples via its tunable prompt (see Figure 1b), and is tasked with generating a prompt ${p'}_{\text{tunable}}$ for the LLM-based agent.

[1]In Voyager, these prompts are hand-engineered and not automatically tuned.



(a) Schematic of LLM-based agent. $p_{tunable}$ can be updated from feedback and/or previous experiences via our sequential optimization.

(b) LLM-Optimizer is a specific LLM-based agent that incorporates previous experiences into the tunable prompt p_{tunable} of the agent.

3 Optimizing LLM-based Agents

Define an LLM agent as $\pi : P_{\text{task}} \times P_{\text{tunable}} \to O$. The distribution O is defined by p_{tunable} alone, which we can regard as the parameter of the LLM agent. The optimization problem we need to solve is to find $p^{\star}_{\text{tunable}} := \arg\max_{p_{\text{tunable}}} \mathbb{E}_o \left[r \mid \pi(p_{\text{task}}, p_{\text{tunable}}) \right]$. We can define an optimizer $g : P_{\text{tunable}} \times F \times R \to P_{\text{tunable}}$, where the goal is to find $p^{\star}_{\text{tunable}}$ through a limited number of times that π attempts the task. An optimal optimizer $g^{\star}$ can find $p^{\star}_{\text{tunable}}$ with the fewest amount of attempts. Different from the reasoning task setup, the output is defined as a distribution even for a single task p_{task}, and oftentimes, we do not know the distribution $o^{\star}$ that can obtain the highest expected reward.

3.1 Fundamentals of LLM Optimization

The most common approach for optimization is through an iterative solver that improves monotonically. However, in order to construct an iterative solver, the optimization problem needs to satisfy a few assumptions. To establish intuitions, we start with numerical optimization in a function approximation-based supervised learning setting. Given a hypothesis h and (x, y), let $\tilde{y} = h_{\theta}(x)$. With a loss function $\ell : X \times Y \times \Theta \to \mathbb{R}$, we can define $L(\theta) = \mathbb{E}_{(x,y)} \left[\ell(\theta, x, y) \right]$. The goal is to find $\theta^{\star} = \arg\min_{\theta} L(\theta)$. To achieve this goal, a valid optimization procedure proposes a new θ for k number of times. They usually consist of two steps:

S1 **Finding Valid Search Direction**: We need to find useful information, such as a descend direction $\Delta\theta^{(k)}$ that can help the update step. The usefulness of the information is tightly coupled with what the update step is.

S2 **Decide Update Rules**: We need to decide how to update θ. A typical update procedure is simply: $\theta^{(k+1)} = \theta^{(k)} + t^{(k)}\Delta\theta$, if $\Delta\theta$ is informative, where t_k is the step size.

If L is convex, then the criteria to determine whether $\Delta\theta$ is informative is quite simple: we can use the gradient of L. From convexity, we know that $\nabla L(\theta^{(k)})^T (L(\theta^{(k+1)}) - \theta^{(k)}) \geq 0$ implies $L(\theta^{(k+1)}) \geq L(\theta^{(k)})$. Then we can set the descend direction $\Delta\theta^{(k)}$ to satisfy $-\nabla L(\theta^{(k)})^T \Delta\theta^{(k)}$. A simple way to satisfy this criterion is let $\Delta\theta^{(k)} := -\nabla L(\Delta\theta^{(k)})$, which is the gradient descent method (GD). However, more complicated update rules can be used, such as backtracking line search [1]. It is also worth noting that we do not always need to satisfy **S2**. For example, in an evolutionary search algorithm, many candidates are proposed and the update rule is simply to keep the candidate with the best score.

Then we can contrast the setting with an LLM optimization problem. If we want to have an iterative descent algorithm to find the optimal prompt for an LLM agent, then we need to consider the following properties:

S1 **Search Direction**: We should obtain useful information, analogous to $\nabla L(\theta)$, to help inform the optimizer on how to update the parameter p_{tunable}.

S2 **Update Parameter**: Unlike the numerical case, where basic algebra can be applied to update parameters, it is unclear whether there is a predefined notion of $\Delta p^{(k)}_{\text{tunable}}$ in text space. This distance $\Delta p^{(k)}_{\text{tunable}}$ in the best case, can be assessed by human intuition over the semantics of the text, in the worst case, can be completely arbitrary.

In order to propose an optimization algorithm using LLM as an optimizer, we must make the following assumptions:

A1 Permissible Search Direction: There exists useful information, which we describe as feedback, f, for an LLM optimizer g such that g can propose a p^{k+1}_{tunable} where $\mathbb{E}_o\left[r \mid \pi(p_{\text{task}}, p^{k+1}_{\text{tunable}})\right] \geq \mathbb{E}_o\left[r \mid \pi(p_{\text{task}}, p^{k}_{\text{tunable}})\right]$.

A2 Valid Update: The LLM optimizer g can modify p_{tunable} based on f, where direction of change: $\Delta p_{\text{tunable}}$ is determined by the information contained in f (i.e., not a random text edit).

In the next few sections, we describe a few possible settings where these assumptions can be satisfied or need not be. We assume **A2** is always satisfied. The first setting we discuss is that it is possible that LLM itself acts as a black-box optimizer. For example, it might obtain a valid search direction by implicitly computing finite differences between inputs and outputs in text space.

LLM Might Implicitly Perform Newton's Method Similar to Newton's Method for using finite difference to approximate gradient, we can hope that LLM can implicitly compute the following function:

$$\nabla R = \lim_{\Delta p^{(k)}_{\text{tunable}} \to 0} \frac{\mathbb{E}_o\left[r \mid \pi(p_{\text{task}}, p^{k}_{\text{tunable}})\right] - \mathbb{E}_o\left[r \mid \pi(p_{\text{task}}, p^{k+1}_{\text{tunable}})\right]}{\Delta p^{(k)}_{\text{tunable}}}$$

If we think this is possible, then the input to the LLM can be tuples of $(p^1_{\text{tunable}}, r_1), ..., (p^k_{\text{tunable}}, r_k)$. Although it is unclear if this is truly the case, this shows that the optimizer needs to retain a history of how past prompts p have changed the reward r.

Feedback Can Be Directional The other possibility, not relying on the black box "magic" of the LLM's internal process, is to hope that somehow a permissible search direction f is given to us from an external source. Humans give directional feedback quite often: "This coffee is too hot for me." or "Can you lower the room temperature?" The first feedback implicitly asks the agent to make a cooler coffee (but not saying exactly how cool it should be). The second feedback asks the agent to turn down the room temperature (but without specifying which temperature to set). Imagine if the agent's action (output space O) for both cases is to write API calls to set temperature; then we know immediately what ΔO should be – keep everything the same, but enter a lower temperature value. After the adjustment, a user might say: "This coffee is now too cold for me." or "I'm freezing!" Making this kind of feedback very similar to the gradient information we get from numerical optimization. This suggests that, in some cases, incorporating feedback (or somehow obtaining directional feedback) could be helpful for the optimization procedure. We should provide LLM optimizer with examples of $(p^1_{\text{tunable}}, f_1, r_1), (p^2_{\text{tunable}}, f_2, r_2), ..., (p^k_{\text{tunable}}, f_k, r_k)$ instead.

Non-directional Feedback There is another type of feedback we can consider. This type of feedback contains useful information but is not directional because they do not directly inform us how to change the input O. For example, feedback like "I can't drink this coffee because the temperature is not quite right." This feedback clearly states that the attribute of "temperature" is important to the user and is not satisfactory. However, it does not tell us whether we should make a coffee that's hotter or colder. Coffee can have many attributes, such as "temperature", "acidity", "roast", "sweetness", or "cream-level". This feedback is more useful than a scalar reward because it *explains* attributes that affect R and allows us to focus more on a single dimension of many attributes.

Reward as Feedback / No Feedback Unlike directional and non-directional feedback, which usually contains information about how to change p_{tunable}, score-based feedback only gives back a numerical value indicating how well p_{tunable} performs. In our setup, this means LLM-Optimizer only observes reward r without f. This is often referred to as the 0th-order feedback.

3.2 Sequential Prompt Optimization

Inspired by the descent method in numerical optimization, we propose an algorithm that aims to satisfy the requirement of descent methods such that we can reach the extremum. We define the following optimization loop with an LLM-based agent π. Agent π with an initial tunable prompt p_{tunable} takes a task description p_{task} from the environment and samples an output o_1. The environment returns a reward r_1 and feedback f_1. An LLM-optimizer stores $(o_1, r_1, f_1, p_{\text{tunable}})$ in a history buffer $\mathbb{H}$. When the buffer becomes large, we subsample H from $\mathbb{H}$. The LLM-optimizer proposes a new tunable parameter p'_{tunable}. We make an explicit decision on whether to replace p_{tunable} with p'_{tunable} based on the reward evaluated on the distribution o and o'. We describe the full procedure in Algorithm 1. We now describe the implementation choices for each component of our iterative solver.

Policy Policy is an LLM that takes in a tunable instruction prompt p_{tunable}, description of task p_{task} and produces an output o. It does not see a history of interaction it did with the environment. This design decision prohibits the policy from controlling its own prompts. It is different from some other existing work. For example, Voyager [10] would allow the policy to see all of its interaction histories (and errors they make). React [12] also allows the policy to see the full error trace. Allowing the policy to see its past errors is a specific design choice on p_{tunable}. We hope to let the optimizer decide what p_{tunable} should be without injecting human prior.

Prompt Proposal We define the prompt proposal module as $\Delta : P_{\text{task}} \times P_{\text{tunable}} \times F \times R \to P_{\text{tunable}}$. This module looks at the task description, past prompts, and the feedback each prompt receives and proposes a new prompt. If the environment provides directional or non-directional feedback, this module should take in F as well. Even though past prompts were included, this module is allowed to produce completely new prompts.

Algorithm 1: Sequential Prompt Optimization

Input: Given s, R, an LLM-based agent π, an LLM-based prompt proposal module Δ, p^0_{tunable}, p_{task}, and K iterations.
Output: p^K_{tunable}

$\mathbb{H} = \emptyset$
for $k \leftarrow 0...K$ **do**
  $o_k, r_k \sim \pi(p_{\text{task}}, p^{(k)}_{\text{tunable}}), R$
  $f_k \sim F$ or $\hat{F}(p_{\text{task}}, o_k, r_k)$
  $H = \texttt{Sample}(\mathbb{H})$
  $p^{(k+1)} = \Delta(H, \{p_{\text{task}}, p^k_{\text{tunable}}, f_k, r_k\})$
  if $\mathbb{E}_o\left[r \mid \pi(p_{task}, p^{k+1}_{tunable})\right] \geq \mathbb{E}_o\left[r \mid \pi(p_{task}, p^{k}_{tunable})\right]$ **then**
    $p^{k+1}_{\text{tunable}} = p^k_{\text{tunable}}$
  end
  $\mathbb{H} = \mathbb{H} \cup \{o_k, r_k, f_k, p^k_{\text{tunable}}\}$
end
return $p^K_{tunable}$

Feedback Synthesizer If numerical feedback is the only type of feedback given, we can design a feedback synthesizing module $\hat{F} : P_{\text{task}} \times O \times R \to F$. It takes in $(o_1, r_1), ..., (o_k, r_k)$ and produces feedback to the prompt proposal module. We designed this module to ask the question, "How should the input be changed to have a greater effect on the objective/output?" We note that we more specifically prompt the LLM to think about the difference in the input space that would impact the difference in output space, different from APO, where they ask the LLM to "give reasons why the prompt could have gotten these examples wrong."

Prompt Selector In order to satisfy the descent method assumption **A1** (permissible search direction), we need to guarantee that p'_{tunable} is an improvement over p_{tunable}. This can be achieved by setting a selection criterion that requires $p^{(k+1)}$ to get a higher reward than $p^{(k)}$. A simple criterion is to improve the average performance over the distribution of O: $\mathbb{E}_o\left[r \mid \pi(p_{\text{task}}, p^{k+1}_{\text{tunable}})\right] \geq \mathbb{E}_o\left[r \mid \pi(p_{\text{task}}, p^{k}_{\text{tunable}})\right]$.

4 Experiments

4.1 Numerical Optimization

We test if LLM is possible to do optimization and what are the necessary ingredients for it to find the optimal solution in an optimization problem. We set up this task because in prompt optimization, it is often hard to know what the optimal prompt or a good search direction is. With a numerical optimization problem, both are well-defined. We use a set of classic optimization problems[2] that require LLMs to find x, a 2-dimensional vector.

[2]https://www.sfu.ca/~ssurjano/optimization.html

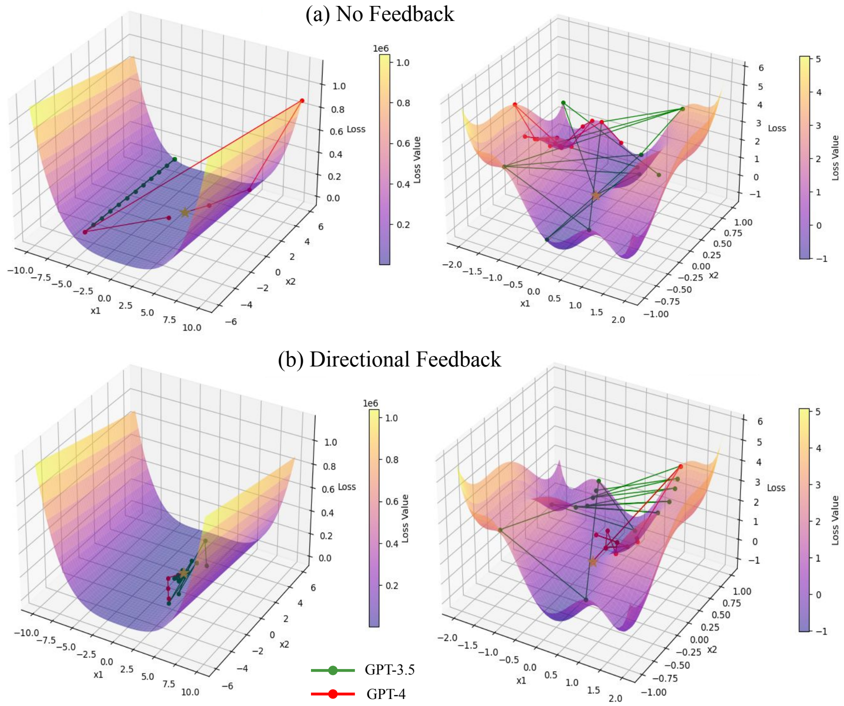


Figure 2: We visualize the optimization trajectory path made by the Optimizer Agent with GPT-3.5 and GPT-4. The loss landscape on the left is the Rosenbrock Function, and on the right is the Six-Hump Camel Function.

1. Task: Given a task description p_{task} and a function J (which is hidden from the LLM), we sample a random starting point $(x_0, J(x_0))$, $x_0 \sim X$. An LLM is asked to produce x to minimize J.
2. Optimizable variable: X. The LLM is asked to output x directly. Here, p_{tunable} is the same as x.
3. Output process O: the output module takes x and directly outputs x, an identity function.
4. Reward R: $R(x) = -J(x)$.
5. Feedback F:
   - Directional Feedback: $\nabla R(x) = \frac{dR}{dx}$, the first-order derivative of the output.
   - Non-directional Feedback: We compare the partial derivatives $\frac{\partial R}{\partial x_1}$ and $\frac{\partial R}{\partial x_2}$, and tell the LLM which dimension of x should be changed to accomplish the task (but without telling LLM in which direction to change).

This experiment does not use the full setup of Algorithm 1. We only test our LLM-based optimizer Δ and our feedback synthesizer $\hat{F}$. We choose four functions: Booth, McCormick, Rosenbrock, and Six-Hump-Camel Function. They were chosen because the optimal x that minimizes these functions is not $[0, 0]$. In our initial experiments, LLM is quick to guess $[0, 0]$, which trivializes the optimization problem.

We define simple regret as $\text{Reg}(\Delta) = |J(x_T) - J(x^\star)|$, where J is the function we try to minimize and T is the number of optimization steps we allow the optimizer Δ to take. We define cumulative regret as $\text{CuReg}(\Delta) = \sum_{t=1}^{T} |J(x_t) - J(x^\star)|$. Intuitively, simple regret corresponds to how close is an optimizer's final answer x_T to the correct answer $x^\star$. Cumulative regret describes how "efficient" is the optimizer at finding the $x^\star$. We compare three models: Δ with **GPT-3.5**, **GPT-4**, and a stochastic gradient descent algorithm (**SGD**) with a small yet fixed learning rate. In the reported results, we run 10 trials and allow Δ to take at most 10 optimization steps.

RQ1 Can an LLM Implicitly Perform Newton's Method, given $(x_1, J(x_1), ..., (x_k, J(x_k))$?

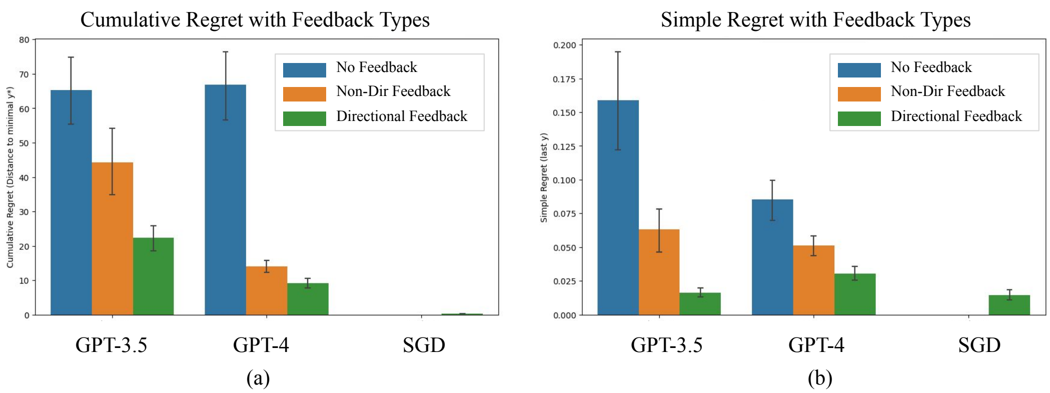


Figure 3: We plot out the average Cumulative Regret and Simple Regret of each condition over 10 trials. Each algorithm is allowed to take 10 steps. We tuned the SGD learning rate slightly to ensure it is not too large or too small. The result is aggregated over 4 loss functions.

From Figure 2, we can see that LLM, as an optimizer, has a rough sense of direction, given a history of past explorations. In Figure 2 (a), we note that in both loss landscapes, although GPT-3.5 often fails to find the minimal point without feedback (green lines), GPT-4 is able to understand the past history and make new proposals of x that incrementally minimizes $J(x)$. This suggests that even though there is no explicit gradient computation, LLM can be asked to "improve" based on a history of observations.

RQ2 Does directional Feedback help the optimization process? Do other types of feedback help as much?

We designed the prompt space for the LLM-based optimizer Δ to insert feedback text right after the observation text and with an additional wording that reads, "You should incorporate the suggestion." Besides this change, the optimizer agent prompt stays the same between no feedback and with feedback conditions. The full prompt is available in the Appendix.

From Figure 2 (b) and Figure 3, we can see that both GPT-3.5 and GPT-4 are able to take advantage of the additional feedback information and improve their search direction. Feedback can help both a weaker model (GPT-3.5) and a strong model (GPT-4). A stronger model can improve more, even if the feedback has less information (see the comparison between Non-directional Feedback and Directional Feedback in Section 3.1).

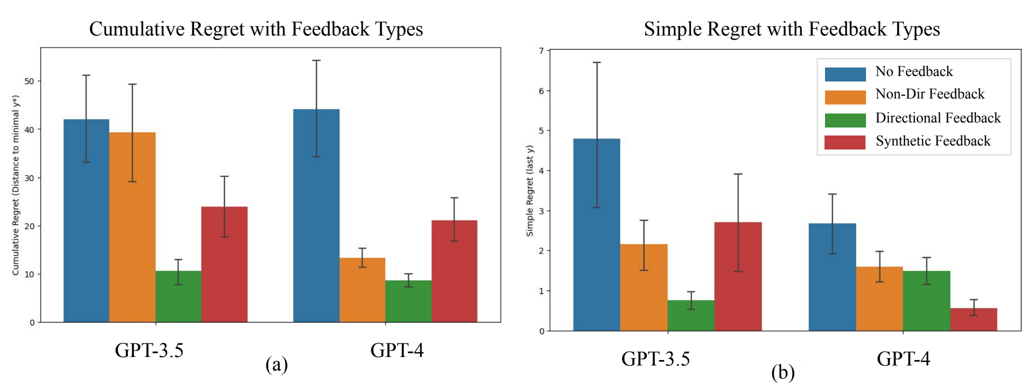


Figure 4: We plot out the average Cumulative Regret and Simple Regret of each condition over 10 trials and compare different feedback types. **Synthetic Feedback** is generated by the same LLM as the optimizer.

Although loss minimization is a challenging task for LLMs, with some amount of feedback, LLMs are able to find a final x that is similar to a classic optimization algorithm like SGD (see Figure 3b) – the simple regret is similar. It is worth noting that GPT-4's final proposed x is not as close to the optimal as GPT-3.5. This is potentially because both models decide their own step size, and we are limiting the optimization horizon to 10 steps.

RQ3 If directional feedback is missing, can we replace it with an LLM module to enhance whatever feedback is available?

Oftentimes, direct and useful feedback might be missing from the environment. In this experiment, we design a feedback synthesizer module (described in Sec 3.2) that can take the output from the model and the reward and try to provide feedback that can improve the next output. Different from methods such as self-reflection, self-criticism, or thinking step-by-step, the feedback synthesizer asks questions similar to "How should I change about x that will result in a larger change in y?", where self-reflection usually asks the model to reflect upon past "mistakes" on what they did wrong.

In Figure 4, we show that we can synthesize feedback from a history of past outputs and rewards that is able to guide the optimizer LLM to find a better solution. Synthesized feedback is not as informative as directional feedback that comes from the environment, but it can easily outperform settings where no feedback is given.

4.2 Poem Generation

Now, we have validated the importance of feedback. We want to validate our optimization setup on a more challenging domain, where now we have to optimize over a prompt that controls how another LLM-based agent produces output. An easy constrained optimization problem to set up is poem generation. A formal poem is a writing assignment that requires the creation of a poem to satisfy some requirements regarding its form. For example, Haiku is a type of formal poem that asks for three lines that form a 5-7-5 syllable pattern. This is a challenging task for both GPT-3.5 and GPT-4, but easy for us to verify whether the generated poem has satisfied the constraint.

1. Task: Generate a poem with a given constraint sampled from a set of constraints.
2. Optimizable variable: P_{tunable}: This is the prompt p_{tunable} for the LLM-based agent that we want to update and optimize.
3. Output process O: the LLM agent takes the prompt p_{tunable} and follows its suggestion and a task description p_{task} to produce a poem o.
4. Reward R: The fraction of lines in the generated poem that satisfy the constraint described by p_{task}. $r \in [0, 1]$.
5. Feedback F:
    - Directional Feedback: We print out the number of syllables in the current line and whether LLM needs to increase or decrease the number of syllables in that line.
    - Non-directional Feedback: We print out how many lines violate the poem writing constraints.

In the following experiment, we use the full setup of Algorithm 1. We allow each agent to take 10 optimization steps. We name our agent **OptAgent**. It produces an instruction that will be sent to the

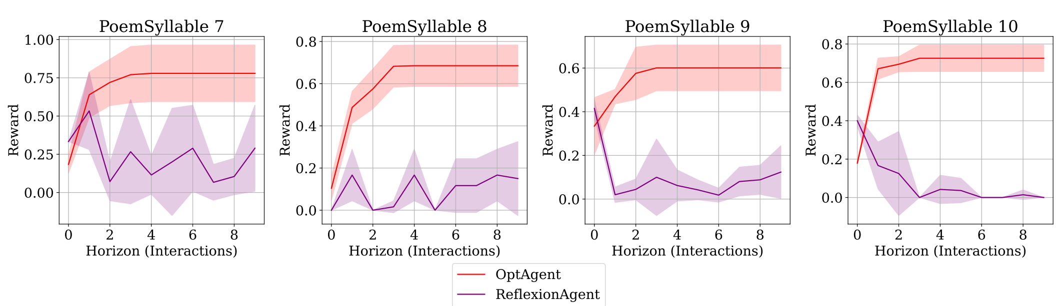


Figure 5: We show the reward for each policy after each round of interaction with the environment. OptAgent (our algorithm) is in red.

poem generation agent to produce a poem. The poem-generation agent will not see the history of mistakes or any other information. We additionally evaluate **Reflexion agent** [8]. We set up four tasks: generating poems that contain 7, 8, 9, or 10 syllables for each line.

We show that in Figure 5, we can reliably select prompts that improve the policy performance for each task. The prompt selection step in our optimization algorithm ensures that the new prompt will improve the performance. Otherwise, it will reject the updated prompt and keep the previous prompt. This differs from the Reflexion Agent in that they are not guaranteed to improve in the next interaction.

5 Conclusion

This paper argues that LLMs can successfully optimize a wide range or entities ranging from from mathematical functions to prompts for textual tasks if provided with directional feedback. We empirically show on challenging numerical optimization scenarios and constrained text generation tasks that utilizing either environment-provided or synthetic feedback is a crucial piece in LLM-based optimization. We emphasize that this is an early work on general LLM-based optimizers. LLMs' potential in this role is still waiting to be realized with improved methods for directional feedback generation.

References

[1] Boyd, S. P. and Vandenberghe, L. (2004). *Convex optimization*. Cambridge university press.

[2] Gur, I., Furuta, H., Huang, A., Safdari, M., Matsuo, Y., Eck, D., and Faust, A. (2023). A real-world webagent with planning, long context understanding, and program synthesis. *arXiv preprint arXiv:2307.12856*.

[3] Ichter, B., Brohan, A., Chebotar, Y., Finn, C., Hausman, K., Herzog, A., Ho, D., Ibarz, J., Irpan, A., Jang, E., Julian, R., Kalashnikov, D., Levine, S., Lu, Y., Parada, C., Rao, K., Sermanet, P., Toshev, A. T., Vanhoucke, V., Xia, F., Xiao, T., Xu, P., Yan, M., Brown, N., Ahn, M., Cortes, O., Sievers, N., Tan, C., Xu, S., Reyes, D., Rettinghouse, J., Quiambao, J., Pastor, P., Luu, L., Lee, K.-H., Kuang, Y., Jesmonth, S., Joshi, N. J., Jeffrey, K., Ruano, R. J., Hsu, J., Gopalakrishnan, K., David, B., Zeng, A., and Fu, C. K. (2023). Do as i can, not as i say: Grounding language in robotic affordances. In *CORL*.

[4] Kojima, T., Gu, S. S., Reid, M., Matsuo, Y., and Iwasawa, Y. (2022). Large language models are zero-shot reasoners. In *NeurIPS*.

[5] Mitchell, M. (1998). *An introduction to genetic algorithms*. MIT press.

[6] Mockus, J. (1998). The application of bayesian methods for seeking the extremum. *Towards global optimization*, 2:117.

[7] Pryzant, R., Iter, D., Li, J., Lee, Y. T., Zhu, C., and Zeng, M. (2023). Automatic prompt optimization with "gradient descent" and beam search. *arXiv preprint arXiv:2305.03495*.

[8] Shinn, N., Cassano, F., Labash, B., Gopinath, A., Narasimhan, K., and Yao, S. (2023). Reflexion: Language agents with verbal reinforcement learning. *arXiv preprint arXiv:2303.11366*.

[9] Sutton, R. S., McAllester, D., Singh, S., and Mansour, Y. (1999). Policy gradient methods for reinforcement learning with function approximation. In *NeurIPS*.

[10] Wang, G., Xie, Y., Jiang, Y., Mandlekar, A., Xiao, C., Zhu, Y., Fan, L., and Anandkumar, A. (2023). Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*.

[11] Yang, C., Wang, X., Lu, Y., Liu, H., Le, Q. V., Zhou, D., and Chen, X. (2023). Large language models as optimizers. *arXiv preprint arXiv:2309.03409*.

[12] Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K. R., and Cao, Y. (2023). React: Synergizing reasoning and acting in language models. In *ICLR*.

A Appendix

A.1 Loss Optimizing Experiment Details

We designed the prompt for two agents. The prompt is written in a Handlebar syntax, where "{{}}" indicate variables to be replaced. A brief guide on this syntax is available here[3].

For the **LLM-Optimizer**:

```
{{#system~}}
You are trying to minimize the output (y) of a function by choosing input (x).
{{~/system}}

{{#user~}}
{{task_description}}

This is what you have previously chosen for x and what the ys were:
{{observation}}

{{feedback}}

You should incorporate the suggestion to output the next x.
Please output the next x that will make this function output the smallest y.
You cannot repeat the same x, doing so will result in a penalty.

Format: x = [x1, x2]
Output:
{{~/user}}
```

For the **feedback synthesizing LLM**:

```
{{#system~}}
You are trying to minimize the output (y) of a function by choosing input (x).
{{~/system}}

{{#user~}}
You are trying to minimize the output (y) of a function by choosing input (x).
You get to observe y once you choose the value of x, where x is a 2-dimensional vector.
This means x = [x1, x2], where x1 and x2 are real numbers.
The goal is to choose x such that y is as small as possible.

Here is a list of x and how it affects y:
{{#each history}}
{{this.action}}
{{this.observation}}
===================
{{~/each}}

For x = [x1, x2]
What are the suggestions you can give to the user to make y smaller?
For example, here are some of the things you can suggest:
- Changing x1 seems to have a bigger effect on y than changing x2.
- Make a larger change on x2
- Increase x1 by 1.2
- Decrease x2 by 0.5
- Try to increase x1 and decrease x2 at the same time
Or any other kind of suggestion. Do not make a suggestion that's the form of a question.
You should only make a one-sentence suggestion that's brief and short.

Suggestion:
{{~/user}}
```

A.2 Poem Experiment Details

For the **LLM-agent** that generates the poem, we use the following prompt:

[3] https://github.com/guidance-ai/guidance

```
{{#system~}}
You are a student and your teacher gives you an assignment to write a poem.
{{~/system}}

{{#user~}}
The assignment is:
{{assignment}}

{{#if exists_intrusction}}
In addition, here are some helpful advice and guidance:
{{instruction}}
{{/if}}
{{~/user}}
```

For the **feedback synthesizer module**, we use this prompt:

```
{{#system~}}
You are a helpful assistant who aims to provide feedback to a student who's writing a poem
according to some instructions.
It is important to let the student know if they did satisfy the instruction or not and why.
{{~/system}}

{{#user~}}
This is the history of past generated poems and how well they did with respect to instructions.

{{#each history}}
Instruction: {{this.observation}}

Poem:
{{this.action}}

Feedback from the teacher:
{{this.feedback}}
---------------
{{~/each}}
{{~/user}}

{{#user~}}
Now, the student writes a new poem.

New instruction: {{observation}}

Poem:
{{action}}

What changes can you make to the poem to help it conform to the instructions?
{{~/user}}

{{#assistant~}}
{{gen 'exp_feedback' temperature=0.7}}
{{~/assistant}}
```

For the **LLM-based optimizer**, we use this prompt:

```
{{#system~}}
You are a helpful assistant that wants to come up with instructions to a student to help them write a poem that is satisfactory to a teacher's assignment.
The student's poem needs to satisfy the requirement of this assignment.
{{~/system}}

{{#user~}}
```

```
This is the history of how you have been helping this student and whether
 ↪  your instructions have succeeded.
Teacher's feedback is the most important feedback, because the student
 ↪  needs to meet the teacher's criteria.
However, another student's feedback can provide helpful information too.

{{#each history}}
The Assignment: "{{this.assignment}}"

Your Instruction:
{{this.prompt}}

Student's Poem:
{{this.action}}

Teacher's Feedback:
{{this.feedback}}

Feedback from another student:
{{this.exp_feedback}}
---------------
{{~/each}}
{{~/user}}

{{#user~}}

Your previous instruction didn't work -- the students didn't write a poem
 ↪  that satisfied the teacher's criteria.
Based on your interaction with the students, can you come up with better
 ↪  instructions that can help this student write a poem that matches the
 ↪  teacher's criteria?
Keep in mind that telling the student what to do step-by-step might be
 ↪  very helpful!
However, you need to be brief and to the point.
{{~/user}}
```