Trace is the New AutoDiff — Unlocking Efficient Optimization of Computational Workflows

Anonymous Author(s)
Affiliation
Address
`email`

Abstract

We study a class of optimization problems motivated by automating the design and update of AI systems like coding assistants, robots, and copilots. We propose an end-to-end optimization framework, Trace, which treats the computational workflow of an AI system as a graph akin to neural networks, based on a generalization of back-propagation. Optimization of computational workflows often involves rich feedback (e.g. console output or user's responses), heterogeneous parameters (e.g. prompts, hyper-parameters, codes), and intricate objectives (beyond maximizing a score). Moreover, its computation graph can change dynamically with the inputs and parameters. We frame a new mathematical setup of iterative optimization, Optimization with Trace Oracle (OPTO), to capture and abstract these properties so as to design optimizers that work across many domains. In OPTO, an optimizer receives an execution trace along with feedback on the computed output. Trace is the tool to implement OPTO in practice: Trace has a Python interface that efficiently converts a computational workflow into an OPTO instance using a PyTorch-like interface. Using Trace, we develop a general-purpose optimizer called OptoPrime that can effectively solve OPTO problems. In empirical studies, we find that OptoPrime is capable of first-order numerical optimization, prompt optimization, hyper-parameter tuning, robot controller design, code debugging, etc., and is often competitive with specialized optimizers for each domain. We believe that Trace, OptoPrime and the OPTO framework will enable the next generation of interactive agents that automatically adapt using various kinds of feedback.

1 Introduction

Computational workflows that integrate large language models (LLMs), machine learning (ML) models, orchestration, retrievers, tools, etc., power many state-of-the-art AI applications [1]: from chatbots [2], coding assistants [3], robots [4], to multi-agent systems [5]. However designing a computational workflow requires laborious engineering because many heterogeneous parameters (e.g. prompts, orchestration code, and ML hyper-parameters) are involved. Moreover, after deployment any erroneous behaviors of the workflow persist unless a developer manually updates it.

We study a class of optimization problems motivated by automating the design and update of computational workflows. Computational workflows produce optimization problems with heterogeneous parameters, rich feedback (e.g. console output and user's verbal responses), and intricate objectives (beyond maximizing a score). Moreover, a workflow can have interdependent steps (e.g. adaptive orchestration, feedback control loops) and/or involve semi-black-box operations whose behavior cannot be succinctly captured(e.g. ML models, simulations). As a result, the structure of the computation may change as the parameters and the inputs of the workflow vary.

Submitted to 38th Conference on Neural Information Processing Systems (NeurIPS 2024). Do not distribute.

Due to its complexity, computational workflow optimization is usually framed as a black-box [6] or algorithm configuration [7] problem, and is tackled by general techniques like Bayesian Optimization [8], Evolutionary Algorithms [9], Reinforcement Learning (RL) [10] using scalar scores as feedback. Recently LLM-based optimizers [11–16] have been proposed to improve efficiency, leveraging the prior of LLMs learned from large pre-training corpora to optimize complex prompts and codes. But one observation of scalar feedback alone does not provide an improvement signal, so these algorithms are very inefficient when the parameter space is large (e.g. codes or natural language prompts). Most of them still use scalar feedback and the workflows contains only a single component (e.g. one LLM call). See Appendix A for discussion on related work.

1.1 Toward Efficient End-to-End Optimization of Computational Workflows

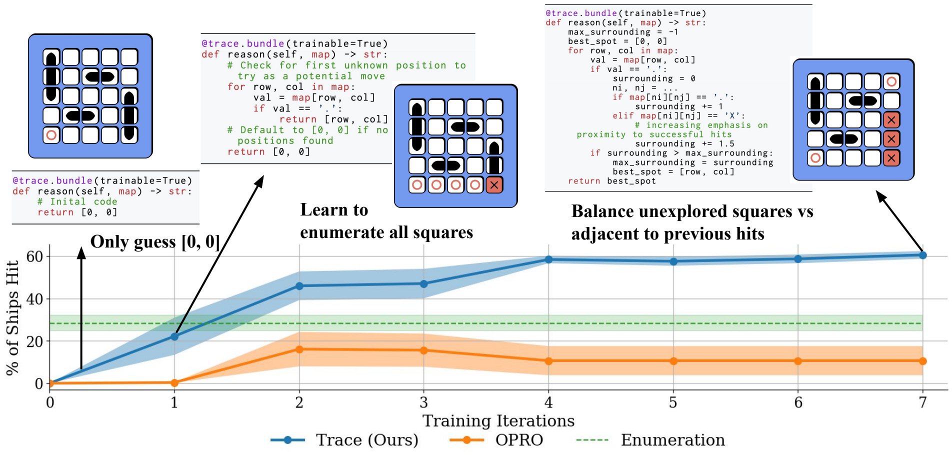
We take an end-to-end approach to computational workflow optimization, inspired by back-propagation [17]. AutoDiff frameworks [18, 19] have scaled back-propagation to optimize differentiable workflows (i.e. neural networks) with billions of parameters. We extend the idea of AutoDiff and design *Trace* for jointly optimizing *all* parameters in general computational workflows.

Trace treats a general computational workflow as a computational graph like a neural network, where nodes are either inputs or parameters (prompts, codes, etc.) or the results of computation steps, and directional edges denote how nodes are created from others. However, instead of gradients, Trace propagates the *execution trace* of a workflow (recording the intermediate computed results and how they are used to create the output). We show that propagating the execution trace subsumes back-propagation for differentiable workflows, and remains applicable even for non-differentiable workflows. Viewing a workflow as a computational graph and then using its execution trace is standard practice for software engineering; for instance, human developers use such traces to debug distributed systems [20]. Our novel insight is that traces also unlock efficient *self-adapting* workflows, because they can provide information to automatically correct heterogenous parameters end-to-end.

1.2 Example of Trace in Action

Trace uses an API inspired by PyTorch [19]. To use it, a user declares the parameters needed to be optimized using a `trainable` flag, decorates the workflow with `node` and `bundle` wrappers, and runs a Trace optimizer; just like how they would declare and train neural networks.

Consider building an AI agent for the Battleship game (Fig. 1). The agent's policy (Fig. 2a) has two components (`reason` and `act`) which are chained together to react to different board configurations. The Battleship environment provides feedback (binary reward) if the agent's action hit the hidden ships, and the goal is to hit all hidden ships as fast as possible. Consider how a human programmer might approach the problem. They may run the policy and change the code based on the observed feedbacks. They may rewrite the code a few times to try different heuristics to solve this problem. They will fix any execution errors (e.g. out-of-bounds exceptions) by using stacktraces.



```
@trace.bundle(trainable=True)
def reason(self, map) -> str:
    # Inital code
    return [0, 0]
```

```
@trace.bundle(trainable=True)
def reason(self, map) -> str:
    # Check for first unknown position to
      try as a potential move
    for row, col in map:
        val = map[row, col]
        if val == '.':
            return [row, col]
    # Default to [0, 0] if no
      positions found
    return [0, 0]
```

```
@trace.bundle(trainable=True)
def reason(self, map) -> str:
    max_surrounding = -1
    best_spot = [0, 0]
    for row, col in map:
        val = map[row, col]
        if val == '.':
            surrounding = 0
            ni, nj = ...
            if map[ni][nj] == '.':
                surrounding += 1
            elif map[ni][nj] == 'X':
                # increasing emphasis on
  proximity to successful hits
                surrounding += 1.5
        if surrounding > max_surrounding:
            max_surrounding = surrounding
            best_spot = [row, col]
    return best_spot
```

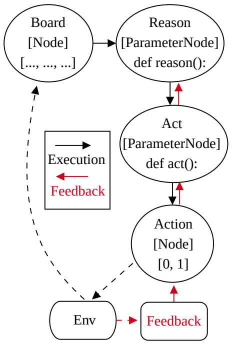
Figure 1: Learning Example in Battleship: An agent playing Battleship must intelligently place a shot on the board. Trace automatically optimizes heterogeneous parameters (e.g. multiple codes) to implement the agent's policy. The `reason()` parameter contains an enumeration heuristic after 2 optimization iterations, and later updates to a balanced explore-exploit strategy. Means and standard errors are computed over 10 random seeds.

```python
class Policy(trace.Module):

 def forward(self, map):
   plan = self.reason(map)
   output = self.act(map, plan)
   return output

 @trace.bundle(trainable=True)
 def reason(self, map) -> str:
   """
   Initial code
   """
   return [0, 0]

 @trace.bundle(trainable=True)
 def act(self, map, plan):
   """
   Given a map and a plan, select a
     target coordinate in a game.
   """
   return [0, 0]
```

(a) We write a trainable policy in Python using Trace operators.

```python
policy = Policy()
params = policy.parameters()
optimizer = trace.Optimizer(params)

env = gym.make('Battleship-v0')
board = env.reset()
done, feedback = False, None
while not done:
  # Forward pass
  try:
    target = policy(board)
    board, feedback, done =
      env.step(target.data)
  except TraceExecutionError as e:
    feedback = str(e)
    target = e.exception_node
  # Backward pass and update
  optimizer.zero_feedback()
  optimizer.backward(target, feedback)
  optimizer.step()
```

(b) We then use PyTorch-like optimization syntax to train the policy.



(c) Trace automatically records execution DAG.

Figure 2: "Complete" Python Code of the Battleship Example. To build a self-adapting agent with Trace, we only need to annotate some empty functions (`reason`, `act`) and set up an optimizer following PyTorch semantics. Trace then builds a DAG as the workflow executes and updates the parameters (see Fig. 1 for the result).

Our Trace framework accomplishes the programmer's goal automatically without adding complexity to the Python code. The user declares `reason` and `act` as trainable (Fig. 2a) and then runs the agent in a PyTorch-like training loop (Fig. 2b). During the execution, Trace records a directed acyclic graph (DAG) (Fig. 2c) and uses it to compute the execution trace for optimization. Trace also automatically catches errors (e.g., syntax/semantic errors) and can use them as feedback. In Fig. 1, we show what the agent learns as Trace optimizes[1] its policy, where the learned policy is evaluated on new randomly generated games. With binary feedback and less than 7 tries, the agent can quickly improve its performance and learn strategies that are increasingly complex. We highlight that Fig. 2a and Fig. 2b are the *full* Python code used to program this efficiently self-adapting agent. Remarkably, there is no mention of Battleship nor details on how the functions `reason` and `act` should behave or adapt in Fig. 2a. The Trace optimizer figures out all the details dynamically as the computational graph unfolds and the feedback on the output is observed. Beyond code as parameters in this example, we also have experiments in Section 5 where prompts and other heterogenous parameters are optimized.

1.3 A New World of Mathematical Optimization

The design of Trace is based on a new mathematical setup of iterative optimization, which we call Optimization with Trace Oracle (OPTO). In OPTO, an optimizer selects parameters and receives a computational graph as well as feedback on the computed output. Trace is a tool to efficiently convert the optimization of computational workflows into OPTO problems in practice.

We argue that framing computational workflow optimization as OPTO can lead to faster convergence than a black-box approach. We present a constructive proof: We design a general-purpose efficient OPTO optimizer called OptoPrime. OptoPrime turns OPTO to a sequence of pseudo-algorithm problems. In each iteration of OPTO, we format the execution trace and output feedback as a pseudo-algorithm question and present it to an LLM for solution (GPT-4 using a ReAct-CoT prompt listed in Appendix G). In experiments, we apply OptoPrime to many disparate applications like prompt optimization, first-order numerical optimization, hyper-parameter tuning, and robot controller design. We find that the general purpose OptoPrime is competitive with specialized optimizers for each domain, e.g. achieving 10% higher accuracy on BigBenchHard [21] when optimizing a DSPy [22] program compared to their hand-designed optimizer.

Working together, Trace, OPTO and OptoPrime provide the first tractable algorithm for optimizing general computational workflows. The Trace framework *a)* leverages the graph structure of a workflow and *b)* can incorporate rich output feedback beyond scores (such as natural language or error messages), extending the concept of AutoDiff to complicated, non-differentiable computational workflows. With Trace, we conjecture that "training deep agent networks" (which fluidly mix computation of tensors, LLMs, and other programmable tools) will soon be possible.

---

[1] We use a new general-purpose LLM-based optimizer OptoPrime that we detail in Section 4.

2 Optimization with Trace Oracle

OPTO is the foundation of Trace. In this section, we define this graph-based abstraction of iterative optimization and discuss how OPTO covers various computational workflow optimization problems.

Preliminary We review the definition of a computational graph (see Fig. 2c). A computational graph g is a DAG, where a node represents an object (such as tensors, strings, etc.) and an edge denotes an input-output relationship. We call a node without parents a root and a node without children a leaf, which are the inputs and outputs of the computational graph. In the context of optimization, some inputs are marked as trainable *parameters*, which are denoted as $\{X_\theta\}$. For a node X, its parents are the inputs to an operator that creates X. The descendents of node X are those that can be reached from X following the directed edges; the ancestors are defined conversely. Without loss of generality, we suppose that all computational operators have a unitary output[2]. In this way, we can associate the operator that creates the child node with the child node, and the full computation can be represented compactly as a DAG without explicitly representing the operators.

2.1 Problem Definition of OPTO

OPTO is an *abstract* setup of iterative computational workflow optimization. An OPTO problem instance is defined by a tuple $(\Theta, \omega, \mathcal{T})$, where Θ is the parameter space, ω is the context of the problem, and $\mathcal{T}$ is a Trace Oracle. In each iteration, the optimizer selects a parameter $\theta \in \Theta$, which can be heterogeneous. Then the Trace Oracle $\mathcal{T}$ returns a *trace feedback*, denoted as $\tau = (f, g)$, where g is the execution trace represented as a DAG (where X_θ are contained in the root nodes of g), and f is the feedback provided to exactly one of the output nodes of g. Finally, the optimizer uses the trace feedback τ to update the parameter according to the context ω and proceeds to the next iteration, as shown in Fig. 3.

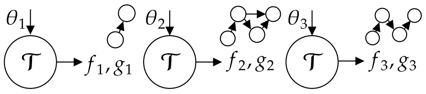


Figure 3: Iterations of OPTO. When $\theta \in \Theta$ is selected, the Trace Oracle $\mathcal{T}$ returns trace feedback $\tau = (f, g)$, where g is a computational graph using θ as an input and f is the feedback given to the output of g.

In OPTO, the output feedback f is generic, e.g. can be scores, gradients, hints/explanation expressed in natural language, and console messages. The context ω provides invariant information to interpret the output feedback f as well as any known side-information, e.g. desired properties of the parameters. The context ω is fixed for an OPTO problem instance (similar to an instruction, or a problem definition), whereas the output feedback f can change with the parameter $\theta \in \Theta$ and the resulting computation g. For example, ω may be "Minimize a loss function" and f is a loss. Alternatively, ω can be open-ended, like "Follow the feedback" and f describes how an output should be changed. In Section 3.1, we discuss how to define the context and output feedback when constructing OPTO problems in practice. In this paper, we focus on OPTO problems where f and ω can be expressed compactly in text. This covers a wide range of problems [23], including those with scalar feedback.

OPTO differs from a black-box setup in that the execution trace g shows the computational path toward the output, which provides information to construct a parameter update direction from f and ω. In the minimization example above, when the execution trace g is missing, it is unclear how the parameter can be improved given only a point evaluation of f. On the other hand, with g, an update direction (e.g., a gradient) can be efficiently derived. We highlight that the structure of the computational graph g returned by the Trace Oracle $\mathcal{T}$ can be different each iteration (as in Fig. 3) because the workflow can change with different inputs and parameters.

To ground the OPTO setup, we show how OPTO is related to some existing problems with examples. We discuss other examples like hyperparameter tuning and multi-agent systems in Appendix C.

Example 1 (Neural network with back-propagation)**.** The parameters are the weights. g is the neural computational graph and f is the loss. An example context ω can be "Minimize loss". The back-propagation algorithm is embedded in the OPTO optimizer. For example, an OPTO optimizer can use τ to compute the propagated gradient at each parameter, and apply a gradient descent update.

Example 2 (RL)**.** The parameters are the policy. g is the trajectory (of states, actions, rewards) resulting from running the policy in a Markov decision process; that is, g documents the graphical model of how an action generated by the policy, applied to the transition dynamics which then returns the observation and reward, etc. f can be the termination signal or a success flag. ω can be "Maximize return" or "Maximize success".

---

[2] A multi-output operator can always be modeled by a single-output operator and single-output indexers.

Example 3 (Prompt Optimization of an LLM Agent)**.** The parameters are the prompt of an LLM workflow. g is the computational graph of the agent and f is the feedback about the agent's behavior (which can be scores or natural language). ω can be "Maximize score" or "Follow the feedback".

3 Trace: The New AutoDiff

We design a framework, Trace, to bring OPTO from an abstract concept to practical reality. Trace provides a light-weight Python tool to implement the Trace Oracle of OPTO when optimizing computational workflows. This is done using a set of wrappers (`node` and `bundle`). Through the OPTO framing, Trace separates the design of optimizers and domain-specific components so that optimizers can be built to simultaneously work across multiple workflows and domains.

Design of Trace Trace is based on two primitives:

- `node` is the wrapper of Python objects. When wrapped, a Python object is registered as a unique node in the global graph of Trace. A node can be set `trainable`, which would make the node a parameter in OPTO. In addition, when using `node` to declare a parameter, one can also describe constraints (in natural language) that the parameter should obey.
- `bundle` is the decorator to turn Python methods into operators. When a function is decorated, its docstring and source code are recorded as the definition of the operator; this information is akin to Jacobians and Adjoints in AutoDiff. Moreover, functions decorated by `bundle` can be set `trainable` as well, which means that the code of the decorated method becomes a parameter.

For any workflow, using Trace involves the following steps (see Fig. 2). First, the user declares the workflow's parameters using `node` and `bundle`, and also defines the workflow's conceptual blocks as operators in the computational graph using `bundle`. Then the user creates an OPTO optimizer (such as OptoPrime as detailed in Section 4), and optionally provides the context ω for the problem. (A default context ω of OptoPrime is "Follow the feedback"). In addition, the user defines a mechanism to provide feedback to the computed result (e.g. scores, natural language suggestions, etc.), in analogy to defining a loss function in neural network training.

Then Trace repeats the following automatically: *1)* Execute the decorated workflow. As it runs, a DAG is built in the backend, logging the computed results and their connections. *2)* Initiate the propagation of the output feedback to the parameters by calling `backward`. (Any execution error is also treated as feedback; see Appendix D.) Internally, Trace extracts the minimal subgraph g connecting the parameters and the output and sends the OPTO optimizer the trace feedback $\tau = (f, g)$. *3)* Call the OPTO optimizer's `step` method to update the parameters.

Trace overloads common Python methods. Therefore, for simpler problems, once the parameters are declared, a workflow code can be optimized directly. For complicated ones, users need to decorate their workflow blocks with `bundle`. The design of `bundle` allows tracing most Python codes, except for those modifying the content of an object reference in place or involve a function recursively calling itself. Such a case can be avoided by duplicating the object first and applying the modification to the copied object, similar to how a recurrent neural network is implemented.

3.1 Guidelines for Constructing OPTO Problems with Trace

There are many ways to represent a computational workflow as a computational graph. In one extreme, the entire computation process is abstracted into one big operator. At the another extreme, every low-level computation is also an operator in the graph. In Trace, the level of abstraction is decided by how `bundle` is applied, as all operations underneath `bundle` are abstracted as one operator summarized by the docstring of that decorated codeblock. Different choices trade-off the complexity of the overall graph and the description needed for each operator. Abstracting everything into a single operator makes the graph simple but requires more descriptions to faithfully capture the workflow. On the other hand, not all details matter in optimization, so exposing every low-level operator in the graph can make it unnecessarily cluttered. Ultimately, the best representation is subjective and depends on the application and OPTO optimizer at hand. This problem we believe is similar to the design of neural network architectures. In this paper, we suggest defining the operators by roughly mimicking the white-board system diagram of the computational workflow. We find that this level of abstraction in our experiments strikes a good balance between the ease of documenting the behavior of each operator and the complexity of the resulting graph.

Apart from architecture design, another under-specified question is what information goes into the context ω versus the description of each operator? For a *single* problem, there is no difference in principle; one can choose to provide details of all operators in g through the context ω. However, this will require manually crafting a context for every workflow. We suggest instead providing a description of the operators when they are defined using `bundle`. Then Trace will automatically generate the workflow-specific information while the same context ω is shared across *many* workflows.

3.2 Backward Step: Implementing the Trace Oracle

Trace uses a recursive graph traversal algorithm (Algorithm 1) to propagate feedback in the reversed topological ordering. By using different propagators, Algorithm 1 can implement various forward-backward schemes including back-propagation.[3] We propose a general propagator, Minimal Subgraph Propagator (MSP), in Algorithm 2. MSP propagates the trace feedback $\tau = (f, g)$, where the computational graph g is implemented as a priority queue. Running Algorithm 1 with MSP (Algorithm 2) together implements the Trace Oracle of OPTO, which extracts the *minimal subgraph*[4] connecting the parameters and an output. Appendix E proves the following theorems:

Theorem 1. *For a graph with N nodes and maximum degree W, Algorithms 1 and 2 have time complexity $O(WN^2 \log N)$ and space complexity $O(WN)$.*

By contrast, back-propagation has a time and a space complexities of $O(Nd^2)$ and $O(d)$, where d is the maximal dimension of tensors. The difference is because in the most general setting of computational graphs and feedback, the propagated feedback (no matter how it is represented) does not have a constant size and needs full information of the subgraph.

Algorithm 1 Backward Message Passing

```
Input: Node $output$, feedback $f$, propagator $P$
$\tau \leftarrow P$.init($f$)
$output$.add_feedback("User", $\tau$)
$queue \leftarrow$ MinHeap([$output$])
while $queue$ is not empty do
    $node \leftarrow queue$.pop()
    $feedback \leftarrow P$.propagate($node$)
    for $parent$ in $node$.parents do
        $\tau \leftarrow feedback[parent]$
        $parent$.add_feedback($node, \tau$)
        if $parent \notin queue$ then
            $queue$.push($parent$)
```

Algorithm 2 Minimal Subgraph Propagator

```
Input: A child node $node$
    // The pseudo code implements propagate.
    // init(f) returns $(f, \{\})$.
$g \leftarrow \{node\} \bigcup \{parent$ in $node$.parents$\}$
for $(f_i, g_i)$ in $node$.feedback do
    $g \leftarrow g \bigcup g_i$
    $f \leftarrow f_i$        // all $f_i$ are the same.
return $\{p : (f, g)$ for $p$ in $node$.parents$\}$
```

Theorem 2. *For generic computational graphs of N nodes, in the worst case, the propagated feedback needs a description length $\Omega(N)$ to construct an improvement direction.*

Despite the worst case complexity of MSP, in practice the difference is negligible. Since MSP only involves merging priority queues of references, most actual computation happens in the forward pass (and also the optimizer's `step` method). For very large problems with thousands of nodes in the minimal subgraph, we anticipate that computational issues of MSP could arise.

4 Design of the First OPTO Optimizer

We introduce an LLM-based optimization algorithm OptoPrime for OPTO problem. Its name indicates that we believe this is one of many possible optimization algorithms for this problem and there is still a large space for identifying efficient optimization methods for OPTO.

Subgraph Representation One core challenge of designing an LLM-based OPTO optimizer is how to represent the execution trace subgraph g (which can involve various graph structures and heterogenous data) to LLMs, in a way that LLMs can understand and reason about the downstream effects of parameter update. We leverage the LLMs' remarkable coding and debugging ability [3]. We present the trace feedback computed by Trace as a pseudo-algorithm problem: we represent the subgraph g as a report of codes with info about the computed values and descriptions of functions involved in g. Based on this report, we ask the LLM to update the parameters in g. Fig. 4 shows an

---

[3] In back-propagation, the message is the gradient ∇_i and the `propagate` function returns $J_i^\top \sum_j \nabla_j$ to its ith parent, where J_i is the Jacobian to the ith parent and and the ∇_j gradient received from the jth child.

[4] The *minimal subgraph* $g_{\mathcal{X} \to Y}$ connecting nodes $\mathcal{X}$ and a node Y is defined as $g_{\mathcal{X},Y} := \mathcal{X} \bigcup \{Y\} \bigcup \{Z | Z \in \text{ancestors}(Y), Z \in \text{descendants}(X), X \in \mathcal{X}\}$.

example of such a report, which is generated by merging the minimal subgraphs from child nodes of the parameter nodes. It is crucial to note that even though the lines look like an actual program, it is not the real program itself but the computational graph defined by `bundle` of Trace (see Section 3.1).

Parameter Update We prompt the LLM with a ReAct-CoT style prompt (listed in Appendix G.2) in one query, asking it to generate reasoning of the graph, an answer, and finally a suggestion on the parameter changes. If the suggestion can be extracted from the LLM's response, we update in-place the parameters.

Optimization Memory OptoPrime optimizes most workflows reasonably well using just instantaneous trace feedback, but it can run into issues when single output feedback is not informative enough (e.g., the output feedback is rewards but the workflow's description doesn't tell how the rewards are generated). For robustness, we have a basic memory module in OptoPrime which tracks the past parameter-feedback pairs and use them as in-context examples. See Appendix G for prompt details.

```
#Code:
a = bar(x)
y = add(b, a)
z = mul(a, y)
#Definitions:
[mul] This is a multiply operator.
[add] This is an add operator.
[bar] This is a method that does
     negative scaling.
#Inputs:
b=1.0
#Others:
a=2.0
y=3.0
#Output
z=6.0
#Variable
x=-1.0
#Feedback:
Output should be larger.
```

Figure 4: An example pseudo-code report generated by Trace for a program of `x = Node(-1.0); z = bar(x) * (bar(x)+1)` and the objective of $\max_x z$.

5 Experiments

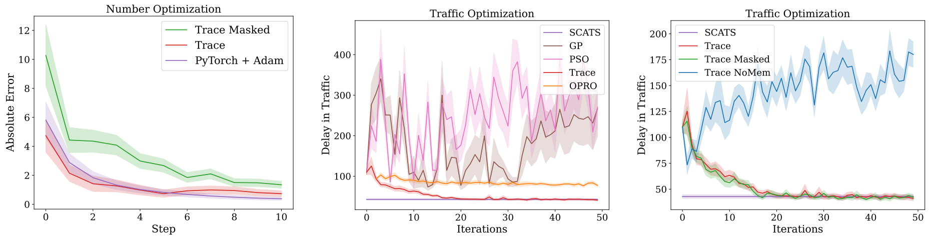
We evaluate the Trace framework with OptoPrime. We implement the state-of-the-art LLM optimizer OPRO [13] as a baseline, which does not use the execution trace but relies on the memory of parameter and feedback pairs. For all experiments, we use GPT-4-0125-Preview. We run the experiments on a standard PC with 16 GB RAM, and Trace introduced no measurable overhead on executing the workflow. In the rest of this section, we will simply denote as Trace+OptoPrime as Trace.

5.1 Validating with Numerical Optimization

First, we want to validate if OptoPrime can solve classical differentiable optimization problems, since they are a special case of OPTO. Consider the problem of $\min_x |h(x) - y^*|$ for a target y^*. We construct a synthetic task environment that randomly creates y^* and the computational graph of h with arbitrarily complex connections between numerical variables (see Appendix B.2 for details). We evaluate OPTO (denoted as Trace) and a variant where the optimizer does not see the graph (Trace Masked); the output feedback is"The output should be <larger/smaller>". We compare their performance with PyTorch's implementation of the Adam optimizer [24]. We run 30 trials over different randomly generated problems. All methods see the same randomness. On average, Trace is able to match the best-in-class first-order gradient optimizer; on the other hand, without access to the full computational graph, the optimizer alone struggles to find y^* (Figure 5a).

5.2 Tuning Hyperparameters to Orchestrate Complex Systems

We tested Trace in a traffic control problem which is an instance of hyper-parameter tuning. We used UXSim [25] to simulate traffic at a four-way intersection, where the trainable parameters are 2 integers in $[15, 90]$, which are the green light duration for each direction of traffic flow. The feedback is the estimated delay experienced by all vehicles due to intersections, and the goal of an optimizer is to minimize the delay using the fewest number of traffic simulations. To this end, this optimizer must find the right trade-off for temporally distributed and variable demands. In Fig. 5 we report the performance of a SOTA heuristic from the traffic control literature, SCATS [26] as well as two black-box optimization techniques: Gaussian Process Minimization (GP) [8] and Particle Swarm Optimization (PSO) [27]. All methods use the same starting parameters. We report further details in Appendix B.3. GP and PSO appear bad because 50 iterations are insufficient for their convergence; given enough iterations, both will eventually perform well. Trace is quickly competitive with the SCATS heuristic, whereas OPRO is not. Moreover, we find that memory is crucial for Trace to perform well for this task. But we note that Trace consumes extra overhead compared to other methods, since Trace has to materialize the resulting computation graph and query an LLM effectively with a longer prompt than that of OPRO.



(a) Numerical Optimization **(b)** Trace vs. Other optimizers. **(c)** Trace ablations

Figure 5: Numerical Optimization and Traffic Optimization

5.3 Unifying Prompts and Functions Optimization

Many LLM agents today, e.g., specified by LangChain [28] and DSPy [22], have many components. These libraries provide optimization tools to optimize a small portion of their workflows, predominantly the prompt that goes into an LLM call. However, for building self-adapting agents that can modify their own behavior, only allowing the change to one part of a workflow but not others seems limiting. In this experiment, we test Trace's ability in joint prompt optimization and code generation. Specifically, we optimize a given DSPy-based LLM agent and tunes its three components: the meta-prompt `prompt_template`, a function `create_prompt` that modifies the prompt with the current question, and a function `extract_answer` that post-processes the output of an LLM call.

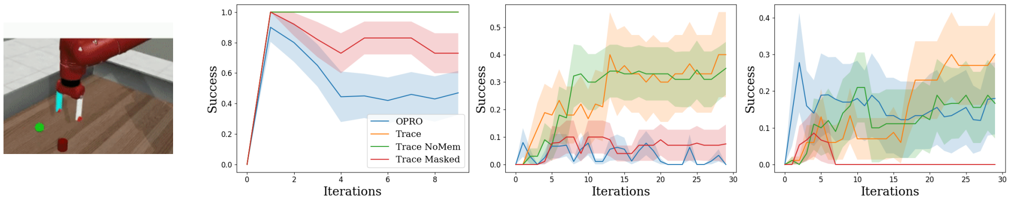
We set up an end-to-end optimization pipeline. Unlike a typical LLM benchmark evaluation, we use an automatic evaluation function to compare the LLM's output with the ground truth, which requires the LLM agent to generate outputs not only with the correct answer but also in the correct format. We use Big-Bench Hard [21] as the problem source (15 examples for training, 5 for validation, and the rest for testing). We compare Trace with DSPy's COPRO module (which optimizes the meta-prompt). In Table 1, we show that Trace is able to optimize a DSPy program beyond what DSPy's COPRO optimizer can offer, especially on algorithmic tasks. This result shows how Trace can concretely improve existing LLM prompting libraries. We show learned codes in Appendix H.

	BBH all (23 tasks)	NLP (12 tasks)	Algorithmic (11 tasks)		BBH all (23 tasks)	NLP (12 tasks)	Algorithmic (11 tasks)
DSPy	41.6	53.8	32.6	DSPy + CoT	70.4	73.7	68.0
DSPy-PO	55.3	69.0	45.2	DSPy-PO + COT	71.6	73.9	70.0
Trace	**59.5**	**70.9**	**51.1**	Trace + CoT	**78.6**	**75.8**	**80.6**

Table 1: End-to-end workflow optimization for an LLM benchmark Big-Bench Hard. `CoT` refers to Chain-of-Thought prompting and `PO` refers to DSPy's own prompt optimizer (COPRO). We use Trace to optimize a DSPy program, starting from the same program and prompt template specified by DSPy.

5.4 Long-Horizon Robot Manipulator Control

We test the ability of Trace to optimize long-horizon workflows with complex dependencies and to "back-propagate through time". We experiment with using Trace to train a controller code (in Python) for a simulated Sawyer robot manipulator. We use the Meta-World environment from LLF-Bench [23] as the simulator and consider three tasks: Reach, Pick-place, and Push. For each task, LLF-Bench provides a task instruction and meaning of the action space, which we use as the context ω of the OPTO problem. The observation is a dict of vectors, indicating the end-effector position, the goal position, the gripper status, etc. The action space is a 4-dimensional vector to control the relative position of the end-effector and the gripper state. In each time step, the LLF-Bench Meta-World simulator returns the observation and natural language feedback to guide the robot. An episode ends if the robot successfully solves the problem or because of time-out. We consider an episodic training setting. The initial condition for all iterations in training is the same. We evaluate the learned policy in terms of success, starting from 10 held-out initial conditions. The task horizon is 10 steps, which is sufficient for task completion, and each training iteration has one rollout. The output feedback in OPTO is success and return. In addition controller code, we also decorate the reset and step functions of the gym environment so that the entire rollout can be traced end-to-end. We compare Trace with



(a) Reach (Test) **(b)** Pick-place (Test) **(c)** Push (Test)

Figure 6: Learning the feedback control policy (code) for a simulated Sawyer manipulator in LLF-Bench Metawrold. In each iteration (x-axis), one episode of rollout is performed and then the policy is updated. Mean and standard error of success rate over 10 seeds are shown.

OPRO; to run ORPO in the streaming OPTO setting, our OPRO implementation only proposes one candidate in each iteration, which is then evaluated and provided with the output feedback.

The experimental results are summarized in Fig. 6. We show learned code in Appendix H. OptoPrime is clearly the top-performing optimizer, especially the version with memory. OPRO is able to solve Reach at the start but its performance degraded over iterations (this instability was mentioned in [13]) and gets similar performance as OptoPrime (without memory) in Push. To validate that the performance of OptoPrime is indeed due to using the execution trace, we include an ablation where we mask out the execution trace, which lead to significant decline in performance and stability. This experiment features the most complex graph structures. The experimental results here are quite impressive, showing that Trace is able to learn a sophisticated control logic in a dozens of interactions, not only working on the training initial conditions but also on the held-out testing ones too. We discuss some limitations in Appendix B.5.

6 Limitations

We highlight that Trace, OPTO and OptoPrime are a first step towards self-adapting workflows and have limitations in their current form. OPTO captures rich feedback, but it is important to specify a solution concept as well the feedback source. We provide guidance for feedback design in Section 3.1 and discuss notions of optimality in Appendix F. Also, Trace cannot convert all computational workflows into OPTO problems; for instance, recursively defined `bundle` operators and distributed/parallel computing workflows are incompatible with the current implementation. Finally, although we demonstrated that OptoPrime can work well with moderate-size graphs, it is not a provably optimal algorithm. The debugging ability and context limits of the LLM used in OptoPrime crucially determines the scale of problems that we can practically address today.

7 Conclusion and Future Work

We created Trace that can convert a computational workflow optimization problem into an OPTO problem, and we demonstrated an efficient OPTO optimizer, OptoPrime. This is just a first step towards a new paradigm of optimization, with exciting avenues for future work.

We connected optimization to an LLM's reasoning capability. Techniques that have been proposed to improve LLM reasoning, e.g. Chain-of-Thought [29], Few-Shot Prompting [30], Tool Use [31], and Multi-Agent Workflows [5] could also help improve OptoPrime or suggest new OPTO optimizers. We conjecture that a hybrid workflow of LLM and search algorithms, with specialized optimization tools can enable a truly general-purpose optimizer. Along the way, we must settle how to delineate the agent vs. the optimizer. How to trade off generality of optimizer vs. crafting side-information in the context ω to achieve task-specific performance is an open question.

In Trace, we chose a specific propagator (MSP), which maximally preserves information for a general computation graph. We can instead specialize it for specific computations, e.g. to accommodate very large graphs. Going a step beyond the memory module we studied in OptoPrime, we anticipate that an optimizer that can reason about how a workflow will behave under counterfactual parameter settings (and not only remember previously tried parameters) can be more efficient than OptoPrime and can enable a divide-and-conquer approach to OPTO.

Finally, in this paper we focused on output feedback and context that can be compactly textualized. We anticipate that computational workflows with rich non-textual contexts and output feedback will also benefit from automatic optimization through appropriate applications of Trace.

References

[1] Matei Zaharia, Omar Khattab, Lingjiao Chen, Jared Quincy Davis, Heather Miller, Chris Potts, James Zou, Michael Carbin, Jonathan Frankle, Naveen Rao, and Ali Ghodsi. The shift from models to compound AI systems. `https://bair.berkeley.edu/blog/2024/02/18/compound-ai-systems/`, 2024.

[2] OpenAI. GPT-4 technical report, 2024.

[3] Albert Ziegler, Eirini Kalliamvakou, X. Alice Li, Andrew Rice, Devon Rifkin, Shawn Simister, Ganesh Sittampalam, and Edward Aftandilian. Productivity assessment of neural code completion. In *SIGPLAN International Symposium on Machine Programming*, page 21–29, 2022.

[4] Jacky Liang, Wenlong Huang, Fei Xia, Peng Xu, Karol Hausman, Brian Ichter, Pete Florence, and Andy Zeng. Code as policies: Language model programs for embodied control. In *ICRA*, pages 9493–9500, 2023.

[5] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, Ahmed Hassan Awadallah, Ryen W White, Doug Burger, and Chi Wang. AutoGen: Enabling next-gen LLM applications via multi-agent conversation, 2023.

[6] Andrew R Conn, Katya Scheinberg, and Luis N Vicente. *Introduction to derivative-free optimization*. SIAM, 2009.

[7] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Learning and Intelligent Optimization*, pages 507–523. Springer, 2011.

[8] Peter I Frazier. Bayesian optimization. *Recent Advances in Optimization and Modeling of Contemporary Problems*, pages 255–278, 2018.

[9] Thomas Bäck and Hans-Paul Schwefel. An overview of evolutionary algorithms for parameter optimization. *Evolutionary computation*, 1(1):1–23, 1993.

[10] Josep Ginebra and Murray K Clayton. Response surface bandits. *Journal of the Royal Statistical Society Series B: Statistical Methodology*, 57(4):771–784, 1995.

[11] Yongchao Zhou, Andrei Ioan Muresanu, Ziwen Han, Keiran Paster, Silviu Pitis, Harris Chan, and Jimmy Ba. Large language models are human-level prompt engineers. In *ICLR*, 2023.

[12] Reid Pryzant, Dan Iter, Jerry Li, Yin Lee, Chenguang Zhu, and Michael Zeng. Automatic prompt optimization with "gradient descent" and beam search. In *EMNLP*, pages 7957–7968, 2023.

[13] Chengrun Yang, Xuezhi Wang, Yifeng Lu, Hanxiao Liu, Quoc V Le, Denny Zhou, and Xinyun Chen. Large language models as optimizers. In *ICLR*, 2024.

[14] Tobias Schnabel and Jennifer Neville. Prompts as programs: A structure-aware approach to efficient compile-time prompt optimization, 2024.

[15] Eric Zelikman, Eliana Lorch, Lester Mackey, and Adam Tauman Kalai. Self-taught optimizer (STOP): Recursively self-improving code generation, 2024.

[16] Allen Nie, Ching-An Cheng, Andrey Kolobov, and Adith Swaminathan. Importance of directional feedback for llm-based optimizers. In *NeurIPS 2023 Foundation Models for Decision Making Workshop*, 2023.

[17] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[18] Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *JMLR*, 18(153):1–43, 2018.

[19] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. PyTorch: an imperative style, high-performance deep learning library. In *NeurIPS*, pages 8026–8037, 2019.

[20] Karthik Nagaraj, Charles Killian, and Jennifer Neville. Structured comparative analysis of systems logs to diagnose performance problems. In *NSDI*, pages 353–366, 2012.

[21] Mirac Suzgun, Nathan Scales, Nathanael Schärli, Sebastian Gehrmann, Yi Tay, Hyung Won Chung, Aakanksha Chowdhery, Quoc Le, Ed Chi, Denny Zhou, et al. Challenging BIG-Bench tasks and whether chain-of-thought can solve them. In *Findings of the ACL*, pages 13003–13051, 2023.

[22] Omar Khattab, Arnav Singhvi, Paridhi Maheshwari, Zhiyuan Zhang, Keshav Santhanam, Sri Vardhamanan, Saiful Haq, Ashutosh Sharma, Thomas T. Joshi, Hanna Moazam, Heather Miller, Matei Zaharia, and Christopher Potts. DSPy: Compiling declarative language model calls into self-improving pipelines, 2023.

[23] Ching-An Cheng, Andrey Kolobov, Dipendra Misra, Allen Nie, and Adith Swaminathan. LLF-Bench: Benchmark for interactive learning from language feedback, 2023.

[24] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[25] Toru Seo. UXsim: An open source macroscopic and mesoscopic traffic simulator in python – a technical overview, 2023.

[26] Courtney Slavin, Wei Feng, Miguel Figliozzi, and Peter Koonce. Statistical study of the impact of adaptive traffic signal control on traffic and transit performance. *Transportation Research Record*, 2356(1):117–126, 2013.

[27] James Kennedy and Russell Eberhart. Particle swarm optimization. In *ICNN*, volume 4, pages 1942–1948, 1995.

[28] LangChain Team. Langchain tracing. `https://blog.langchain.dev/tracing/`, 2023.

[29] 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.

[30] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. In *NeurIPS*, 2020.

[31] Timo Schick, Jane Dwivedi-Yu, Roberto Dessi, Roberta Raileanu, Maria Lomeli, Eric Hambro, Luke Zettlemoyer, Nicola Cancedda, and Thomas Scialom. Toolformer: Language models can teach themselves to use tools. In *NeurIPS*, 2023.

[32] Weiran Yao, Shelby Heinecke, Juan Carlos Niebles, Zhiwei Liu, Yihao Feng, Le Xue, Rithesh R N, Zeyuan Chen, Jianguo Zhang, Devansh Arpit, Ran Xu, Phil L Mui, Huan Wang, Caiming Xiong, and Silvio Savarese. Retroformer: Retrospective large language agents with policy gradient optimization. In *ICLR*, 2024.

[33] Alessandro Sordoni, Xingdi Yuan, Marc-Alexandre Côté, Matheus Pereira, Adam Trischler, Ziang Xiao, Arian Hosseini, Friederike Niedtner, and Nicolas Le Roux. Joint prompt optimization of stacked LLMs using variational inference. In *NeurIPS*, 2023.

[34] Mingchen Zhuge, Wenyi Wang, Louis Kirsch, Francesco Faccio, Dmitrii Khizbullin, and Jürgen Schmidhuber. Language agents as optimizable graphs, 2024.

[35] Ruotian Ma, Xiaolei Wang, Xin Zhou, Jian Li, Nan Du, Tao Gui, Qi Zhang, and Xuanjing Huang. Are large language models good prompt optimizers? *arXiv preprint arXiv:2402.02101*, 2024.

[36] Xinyuan Wang, Chenxi Li, Zhen Wang, Fan Bai, Haotian Luo, Jiayou Zhang, Nebojsa Jojic, Eric P Xing, and Zhiting Hu. Promptagent: Strategic planning with language models enables expert-level prompt optimization. *arXiv preprint arXiv:2310.16427*, 2023.

[37] Xinyu Tang, Xiaolei Wang, Wayne Xin Zhao, Siyuan Lu, Yaliang Li, and Ji-Rong Wen. Unleashing the potential of large language models as prompt optimizers: An analogical analysis with gradient-based model optimizers. *arXiv preprint arXiv:2402.17564*, 2024.

[38] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. Teaching large language models to self-debug. In *ICLR*, 2024.

[39] Zeyuan Ma, Hongshu Guo, Jiacheng Chen, Guojun Peng, Zhiguang Cao, Yining Ma, and Yue-Jiao Gong. LLaMoCo: Instruction tuning of large language models for optimization code generation, 2024.

[40] Tennison Liu, Nicolás Astorga, Nabeel Seedat, and Mihaela van der Schaar. Large language models to enhance bayesian optimization. *arXiv preprint arXiv:2402.03921*, 2024.

[41] Michael R Zhang, Nishkrit Desai, Juhan Bae, Jonathan Lorraine, and Jimmy Ba. Using large language models for hyperparameter optimization. In *NeurIPS 2023 Foundation Models for Decision Making Workshop*, 2023.

[42] Gábor Bartók, Dean P Foster, Dávid Pál, Alexander Rakhlin, and Csaba Szepesvári. Partial monitoring—classification, regret bounds, and algorithms. *Mathematics of Operations Research*, pages 967–997, 2014.

[43] Laurent Hascoet and Mauricio Araya-Polo. Enabling user-driven checkpointing strategies in reverse-mode automatic differentiation. *arXiv preprint cs/0606042*, 2006.

[44] Amirreza Shaban, Ching-An Cheng, Nathan Hatch, and Byron Boots. Truncated back-propagation for bilevel optimization. In *The 22nd International Conference on Artificial Intelligence and Statistics*, pages 1723–1732. PMLR, 2019.

[45] Ching-An Cheng, Jonathan Lee, Ken Goldberg, and Byron Boots. Online learning with continuous variations: Dynamic regret and reductions. In *International Conference on Artificial Intelligence and Statistics*, pages 2218–2228. PMLR, 2020.

[46] Sham Machandranath Kakade. *On the sample complexity of reinforcement learning*. University of London, University College London (United Kingdom), 2003.

A Related Work

Framework for Computational Workflows Frameworks such as LangChain [28], AutoGen [5], DSPy [22] allow composing computational workflows and provide hand-engineered optimizers to tune an LLM's context (i.e. prompt templates, few shot examples, or tool libraries) using scalar feedback with black-box search techniques. They support tracing of the workflow to aid in profiling, debugging and visualization. In contrast, Trace uses tracing for *automatic optimization*, and constructs a different representation of the computational graph which is suited for that purpose. Moreover, Trace is designed to be general-purpose and agnostic to the underlying frameworks of computational workflows users choose. In principle, one can apply Trace to decorate and tune a workflow based on a mix of Autogen, LangChain, DSPy codes. In fact, our experiments in Appendix B use workflows declared using both AutoGen and DSPy.

Optimization of Graphs of LLM Workflows There are multiple efforts to optimize the computational graph of LLM workflows, which is a special case of the OPTO problem. These algorithms focus on optimizing prompts. SAMMO [14] is an example for prompts that uses additional graph structure to make the optimizer efficient. SAMMO represents the prompt parameter itself as a program so as to enable more efficient black-box search through the space of programs. DSPy [22] can optimize directly the prompts or the few-shot examples to include using scalar reward feedback. Retroformer [32] uses another small language model (LM) to provide suggestions/feedback (i.e. changing prompts) to improve the behavior of an actor LLM, where the small LM is tuned by offline RL. Deep Language Networks [33] view all of the prompts in an LLM worflow as tunable parameters and jointly optimizes them. They discovered that optimizing each parameter in isolation instead produces subpar results. [34] frames LLM systems as graph where nodes are operations and edges are messages/connections. (Note that this is different from the DAG used in Trace; here nodes are messages and edges are input-output of operators) and optimizes for the connection on edges (binary variables) by REINFORCE using scalar reward feedback and prompts by LLMs . They optimize each component separately without considering each other; for example, the prompts are optimized individually without considering the graph topology or how they are used down the road. We suspect this approach can be less stable. Their prompt optimization part also does not take output feedback, but simply use an LLM to self-check whether the prompt meets the need of generating desired functions the user specified. In contrast to these works, through the OPTO framing, Trace supports *joint* optimization of all parameters (prompts, hyperparameters, codes) with *rich* feedback, and is agnostic to graph structures (e.g., changing these parameters can dynamically change the graph structure and connections between nodes). Users of Trace are free to specify which parameters they want to automatically optimize via online interactions.

LLM-Optimizers for Prompts and Codes There is a huge and fast growing literature on using LLMs as optimizers to improve prompts [12, 11, 35–37] or codes [38, 15, 39, 35]. Different from the works mentioned above, here the focus has been on an isolated problem (e.g., changing the behavior of a single LLM or improving the code generation in the question-answering format) rather than considering a non-trivial workflow or agent with multiple components like above. They do not consider optimizing prompts or codes as one component of a bigger workflow (e.g. implementing an autonomous agent), which is harder and requires the right credit assignment. Trace can also be applied to optimize trivial OPTO problems where the returned graph has just a single node of the parameter (which are the scenarios considered by these works). Nonetheless, the main focus of this paper is to study how optimization can be done efficiently as the graph becomes nontrivial.

LLM-Optimizers for Hyperparameters Recent works like [40, 41] use LLMs to optimize numerical hyperparameters, as an alternate to Bayesian optimization. Here in the experiments we show that Trace + OptoPrime also can effectively learn hyperparamters, faster than Bayesian optimization. The main difference between Trace and the aforementioned work is the representation of the problem. In Trace, we provide the graph to the LLM-based optimization (through the pseudo-algorithm representation), and we consume rich language feedbacks on the output, both of which accelerates hyper-parameter optimization.

OPTO Related Setups OPTO is a generalization of partial monitoring games [42]. If there exists a latent loss function that the feedback f adheres to (e.g. as in [23]), those OPTO instances can be written as partial monitoring game. However OPTO admits a more general notion of feedback f, and we discuss solution concepts for them in Appendix F. On the other hand, OPTO can be also viewed as a special case of Learning from Language Feedback (LLF) setup defined in [23] with observations as the trace feedback. This is a framing of a *meta* LLF problem. In the LLFBench Meta-World experiments of this paper (Section 5), we show Trace can be used to learn policy for LLF problems grounded to an application too.

AutoDiff and Back-propagation Back-propagation has been shown to be a very effective tool in optimizing differential computational workflows. Our design of Trace is inspired by back-propagation and the ease of use of the AutoDiff framework PyTorch [19]. Nonetheless, we highlight that back-propagation (Backward Mode Differentiation) is not the only AutoDiff algorithm. For example, the gradient can be computed in a forward mode (Forward Mode Differentiation) as well, and there are also techniques of Checkpointing [43] and Truncated Back-Propagation approximation [44] for efficiency. What are the equivalent ideas of these methods for general computational workflows? We think this is an interesting future research direction.

B Experiment Details

B.1 Battleship

We implement a simple battleship game board in Python. The exact code is in the appendix. The game offers a string-based visualization of the board. It randomly places different types of ships on a 2-dimensional board with pre-specified width and height when it initializes. The agent does not see the ship location and has to select a coordinate on the board to hit next. One additional rule of this game is that the agent can go again if their previous coordinate selection (fire) is a hit, not counting as the finish of a turn. In Figure 1, we ran 10 trials, where in each trial, we ran 20 iterations of training. We measure the reward as % of ship squares hit (over all squares occupied by ships). The reward plateaued at 60% because the game has a chance element (heuristics and strategies can only go so far – strategy is only in effect if a hit happens. Otherwise, there is no information about where ships might be).

B.2 Numerical Optimization

Any classical numerical optimization problem can be framed as an OPTO problem. Consider $h(x)$ and a target y^*, in a context ω finding the y^* by changing x; we know the most useful corrective f feedback to change x is the gradient $\nabla_h x$. Similar to Trace, AutoDiff packages like PyTorch's AutoGrad have implemented dynamic graph construction with special classes like `torch.Tensor`. We want to validate whether it is possible to rely on binary text feedback, a graph automatically constructed by Trace, and OptoPrime to update x in the context of minimizing $|y - y^*|$.

We constructed a synthetic task environment where we can create a complex computation graph with arbitrarily complex connections between numerical variables. The focus of this environment is on the complexity of the graph, not on the complexity of the numerical operators. Therefore, we only use one-dimensional input and basic arithmetic operators to create a numerical optimization problem solvable by a first-order optimizer. This environment constructs a computational graph by sampling a number of times. At each time, it will either use a previously computed variable or sample a new variable, and an operation will be sampled to combine them. The optimization task is, for a fixed number of steps, an optimizer needs to output x that minimizes y.

We evaluate the following baseline methods. Basic Agent: a basic LLM agent that simply stores past information of (x_{t-1}, y_{t-1}) in context before choosing the next x_t. OPRO Agent: a basic LLM agent but we implement the state-of-the-art LLM optimizer OPRO [13], which updates the meta-prompt of the basic LLM agent. Torch + Adam: the problem we construct is end-to-end differentiable. Therefore, we simply pass in `torch.Tensor(x)` as input and use Adam optimizer to update. We tune the learning rate slightly and found 1e-1 to work well. We compare two kinds of Trace-based optimizers: Trace, where we allow OptoPrime to read in the entire computation graph before updating x, or Trace Masked, where we hide the computation graph.

We run 30 trials over different computation graphs and start all methods with the same initial x, y^*. We compute the absolute error, which is $|y - y^*|$. On average, Trace is able to match the best-in-class first-order gradient optimizer Adam [24]. It is not entirely surprising that all the other baselines are performing worse due to a lack of access to the computation graph. To our surprise, OPRO, by only accessing the history of input and output, as well as changing the meta-prompt, is able to eventually discover the correct solution. This confirms why there were early signs of success using LLMs for black-box optimization in a simple plug-and-play style. However, OPRO is not an efficient optimizer because it lacks access to the Trace oracle. We show OPRO struggles even more when the computation graph gets more complex.

B.3 Traffic Control

We tested OptoPrime in a traffic control problem which is an instance of hyper-parameter tuning. We used UXSim [25] to simulate traffic at a four-way intersection, where the tunable parameters are the duration of the green lights for each direction of traffic flow. The feedback is a scalar loss calculated by monitoring the flow of a pseudo-random sequence of vehicles arriving at the intersection over a period of 30 minutes. The loss computes an estimate of the delay experienced by all vehicles due to the intersection, as well as variability in this estimate for every link in the network; lower values are better. The goal of an optimizer is to identify values for all of the green light duration so as to minimize the loss using the fewest number of traffic simulations. If the green light duration for a given traffic flow direction is set too low, then vehicles will queue up over time and experience delays, thereby lowering the score for the intersection. However, if the green light duration for a given direction is set too high, vehicles in other directions will queue up and experience delays, thereby lowering the score for the intersection. Hence an optimizer must find the right trade-off for temporally distributed and variable demands.

In Figure 5 we report the performance of a SOTA heuristic from the traffic control literature, SCATS [26] (adapted to this toy setting) as well as two black-box optimization techniques: Gaussian Process Minimization (GP) [8] and Particle Swarm Optimization (PSO) [27]. All methods are initialized to evaluate the same starting parameter. GP and PSO further evaluate 5 random parameters; moreover, if they query a previously evaluated point, that query is replaced by a randomly sampled parameter. GP constructs a surrogate model to mimic the black-box traffic simulation function which maps from parameters to observed score. Then it minimizes a utility function (e.g. the lower confidence bound) using the surrogate model to pick the next parameter to evaluate. PSO on the other hand maintains 5 particles in parameter space, each with a position and velocity. At each iteration of PSO, particles update their positions according to their previous positions and velocity, evaluate the function at the updated positions, and update the velocities of all particles using the observed values. Although GP and PSO are both black-box methods, GP can be thought to replace Trace oracle with instead a smooth differentiable surrogate function; whereas PSO is very different and maintains a candidate set of parameters (can be thought of as conceptually related to OptoPrime with memory).

GP appears to be bad because even when it converged, the exploration heuristic randomly samples parameters rather than pick the converged parameter. PSO appears bad because 10 iterations is insufficient for its convergence. Note that given enough number of iterations, black-box approaches will eventually perform well. Trace is quickly competitive with the SCATS heuristic, whereas OPRO is not. Moreover, we find that memory is crucial for OptoPrime to perform well at this task. Finally, Trace consumes additional overhead compared to black-box methods; beyond the space and time complexity for running the traffic simulation, Trace additionally materializes the computation graph per iteration. Thus it can also be more expensive per LLM call compared to OPRO.

B.4 BigBench-Hard

Perhaps more surprisingly, there are many components that a workflow needs to learn. Some of these components can be the prompt to generate output from an LLM, while other components can be code that needs to further process these outputs. In many workflows today, enabled by LangChain [28] and DSPy [22], only a small part of this workflow, predominantly, the input to an LLM API call, is optimized. These libraries optimize input to an LLM, and human engineers process that input and integrate it into other systems. Indeed, both libraries can enable robust and swift large-scale engineering efforts to build LLM-based software. However, if our goal is to develop self-adapting agents that can modify their own behavior, we should not ignore one of LLM's greatest strengths: code generation. Trace allows us to unify prompt optimization and code generation, which enables the creation of agents capable of fast learning.

```python
import trace

class Predict(trace.Module):
    def __init__(self):
        self.prompt_template = trace.Node("""
        Given ‘question‘, produce the ‘answer‘.
        question: {}
        answer:
        """,
        trainable=True)

    def forward(self, question):
        user_prompt = self.create_prompt(question)
        response = self.call_llm(user_prompt)
        answer = self.extract_answer(question, response)
        return answer

    @trace.bundle(trainable=True)
    def create_prompt(self, question):
        """formulate the prompt with the question"""
        return self.prompt_template.format(question)

    @trace.bundle(trainable=True)
    def extract_answer(self, question, response):
        """Extract the answer out of LLM response"""
        answer = response.split("answer:")[1].strip()
        return answer
```

(a) We write a workflow that prompts an LLM for a question and extracts the answer.

```python
import trace

dataset = load_dataset(link, task)
predict = Predict()
optimizer = trace.Optimizer(predict.parameters())

for ex in dataset:
    # Forward pass
    response = dp.forward(ex[’question’])
    if response == ex[’answer’]:
        feedback = "The answer is correct"
    else:
        feedback = "The answer is incorrect"

    # Backward pass and update
    optimizer.zero_feedback()
    optimizer.backward(response, feedback)
    optimizer.step()
```

(b) The optimizer class takes in any parameter regardless of whether it is code or text. Although the actual optimization implementation can provide different treatments to many input types, the user interface stays consistent.

Figure A.1: LLM-based Workflow Optimization Example.

In this example of an LLM-based workflow (Figure A.1), there are three parameters that are flagged as `trainable` for the optimizer: `prompt_template`, `create_prompt`, and `extract_answer`. Note that two of them require the LLM to generate Python code, and one of them requires the LLM to modify a text. Trace abstracted away the different data types and enabled direct update and optimization of them. Furthermore, a human engineer is often tasked with writing an error-free `extract_answer`. The output of an LLM can be highly stochastic and can often change over time; the code that is used to extract the response of an LLM has to be extremely robust and, therefore, arduous to create. Whenever a major distribution shift happens in the LLM output, this code needs to be rewritten by a human engineer, and it is hard for humans to anticipate all of LLM's output patterns.

We set up the task of end-to-end workflow optimization. Unlike a typical LLM benchmark evaluation, where a lot of effort went into creating the perfect `evaluate(answer, target)` method so that all kinds of LLM outputs were post-processed, cleaned, and formatted to match the ground truth, we choose a simple evaluation function (that extracts a segment or does exact string matching) and place the burden on the workflow itself to figure out how to create the right answer to satisfy the evaluation metric. We choose Big-Bench Hard [21] as our task because it has 23 subtasks and contains both language and algorithmic tasks.

We split each task dataset into training, validation, and test. For Trace and Trace-CoT, we use the first 15 examples for training, 5 examples for validation (picking the best learned workflow), and then evaluate the performance on test examples. DSPy's prompt optimization method does not explicitly require a validation set, therefore, we just used all 20 examples for training. For both, we only optimize for 1 epoch. We either start with the vanilla boilerplate prompt template used by DSPy or we use the slightly sophisticated template used by DSPy's CoT module. Trace optimizes both DSPy's original design and outperform their own optimizer COPRO by 10% on algorithmic tasks.

Big-Bench Hard requires different answer outputs. Out of 23 tasks, 14 tasks require a multiple-choice answer with options provided in the question. 4 of them require yes/no. 1 task requires True/False, while 1 task requires valid/invalid. And the 3 remaining tasks require answers that contain words or numbers. Even though DSPy's meta-prompt optimization is trained on each task individually, the output of LLM to the evaluation method is still not post-processed, resulting in low performances of these tasks. However, Trace can optimize code and LLM prompt jointly to successfully deliver the response expected by an automatic evaluation method.

B.5 LLFBench Meta-World

We test the ability of Trace to optimize long-horizon workflows with complex dependencies. We experiment with using Trace to train controller (python code) for a simulated Sawyer robot ma-

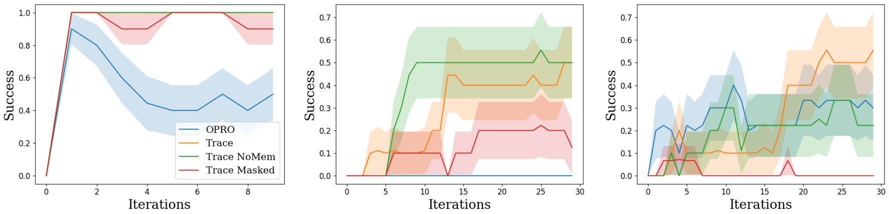
Task Name	DSPy	+CoT	DSPy-PO	+CoT	Trace	+CoT
tracking shuffled objects 7 objects	37.39	90.0	90.43	90.43	37.8	87.8
salient translation error detection	51.3	70.87	51.3	69.57	63.0	70.0
tracking shuffled objects 3 objects	39.13	94.35	97.39	93.91	38.7	96.5
geometric shapes	50.43	62.17	59.13	60.43	49.6	62.2
object counting	0.0	40.0	0.0	74.35	42.2	80.4
word sorting	0.0	0.0	0.0	0.43	84.3	74.3
logical deduction five objects	70.0	80.43	70.0	74.78	48.7	75.7
hyperbaton	74.78	86.52	74.78	88.26	78.3	91.7
sports understanding	0.0	0.0	0.0	0.0	79.6	45.7
logical deduction seven objects	68.7	64.78	68.7	64.78	45.7	69.6
multistep arithmetic two	0.0	93.04	0.0	93.04	94.8	88.7
ruin names	84.35	87.83	84.35	87.83	87.8	90.0
causal judgement	7.78	70.66	74.25	70.66	70.1	54.5
logical deduction three objects	85.22	97.39	85.22	97.83	91.7	97.0
formal fallacies	1.74	81.3	62.17	81.3	73.5	67.8
snarks	86.08	87.34	86.08	87.97	81.6	87.3
boolean expressions	0.0	98.26	64.35	98.26	88.7	96.5
reasoning about colored objects	53.04	91.3	89.13	91.3	91.3	95.7
dyck languages	0.0	8.7	7.83	8.7	26.5	9.6
navigate	0.0	95.65	0.0	97.39	59.6	92.6
disambiguation qa	67.83	66.09	73.91	66.09	75.7	59.1
temporal sequences	99.57	99.13	97.39	99.13	97.8	98.3
web of lies	0.0	0.0	0.0	0.0	49.6	90.4
tracking shuffled 5 objects	37.83	96.09	37.83	96.09	58.3	88.7
penguins in a table	69.84	92.86	97.62	92.86	81.7	91.3
movie recommendation	83.48	76.09	83.48	76.09	81.3	75.7
date understanding	69.13	85.65	69.13	85.65	70.4	85.7

Table A.1: Big Bench-Hard Per-Task Result. Some 0.0 here shown is because DSPy cannot find the clean/stripped output that matches what the automatic evaluation method expects. With additional human engineering, these numbers can improve.

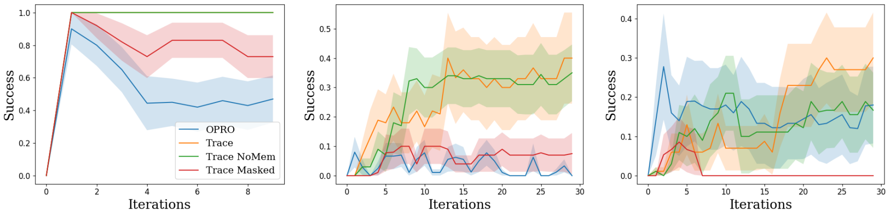
nipulator. We use the Meta-World environment of LLF-Bench [23] as the simulator and consider three tasks reach, pick-place and push. LLF-Bench is a simulated benchmark with gym interface for testing an agent's ability to learn from language feedback. In these LLF-Bench Meta-World tasks, the observation is a dictionary where each field denotes a feature of the state and has a vector value (e.g., the end-effector position, the goal position, the gripper status, etc.). The keys of the observation dictionary can differ for each task. The action space is 4-dimensional, which controls the relative position of the end-effector and the state of the gripper. In each time step, the LLF-Bench Meta-World simulator returns the observation dictionary and natural language feedback to guide the robot (we use the 'a' mode of LLF-Bench, with which the language feedback would contain information about the current performance, explanation of past successes and failures, and suggestions for the next step). An episode ends if the robot successfully solve the problem or because of time-out. For each task, LLF-Bench also provides a task instruction explaining that the task is about controlling a Sawyer robot arm and the meaning of the action space (see [23]). We use that as the context ω of the OPTO problem. We consider an episodic setting. For each experiment (a random seed), we randomly sample an initial configuration. Then for each iteration of optimization, we reset the simulator to that sampled initial configuration and run the robot policy for 10[5] steps or until the episode termination due to success. We compute the sum of rewards and gives the output feedback f in texts in the format of "Success: <true/false> Return: <score>". Note that the initial condition for all iterations within an experiment is the same so that the optimization problem is deterministic. To evaluate the learned policy's performance, for each experiment, we additionally run the learned policy starting from 10 held-out initial conditions, different from the fixed training initial condition. For each training algorithm discussed, we run it with 30 iterations, where each iteration consists of one episode rollout and one update.

---

[5] We set the problem horizon to be 10 steps, as we find the expert policies implemented in LLF-Bench can solve these problems within 10 steps.



(a) Reach (Train) **(b)** Pick-place (Train). **(c)** Push (Train).



(d) Reach (Test) **(e)** Pick-place (Test) **(f)** Push (Test)

Figure A.2: Learning the feedback control policy (code) for a simulated Sawyer manipulator in LLF-Bench Metawrold. In each iteration (x-axis), one episode of rollout is performed and then the policy is updated. Mean and standard error of success rate over 10 seeds are shown.

To optimize the controller with Trace, we declare the control code as the parameter using the `bundle` decorator with `trainable` set to True; the initial control code simply outputs a zero vector $[0, 0, 0, 0]$. We decorate also the reset and the step function of the gym environment, so that the entire rollout of an episode can be traced end-to-end. In our implementation, a prototypical rollout would create a graph with around 30 operations where the controller code parameter is used multiple times. This graph structure is similar to that of running a recurrent neural network. For Trace, we experiment with OptoPrime with and without a memory of size 10. In addition to Trace, we implement the state-of-the-art LLM optimizer OPRO [13] as a baseline. Compared with Trace, OPRO does not use the execution trace information but rely on just memory of parameter and feedback pairs[6] To run OPRO in the OPTO setting, our implementation only proposes a single candidate in each iteration, which is then evaluated and provided with the output feedback. Since in [13] OPRO generates about 10 samples per iteration, so one iteration in [13] is roughly equivalent to 10 iterations here.

The experimental results are summarized in Fig. A.2, where we show the success rates at both the training initial condition as well as the held-out testing initial conditions over 10 seeds. OptoPrime is clearly the top-performing optimizer, especially the version with memory. OPRO is able to solve Reach at the start but its performance degraded over iterations (this instability was observed in [13]) and gets similar performance as OptoPrime (without memory) in Push. To validate that the performance of OptoPrime is indeed due to using the execution trace, we include an ablation where we mask out information in #Inputs, #Others, #Code, #Definition in the LLM context (see Fig. A.3b), which lead to significant degrade in performance and stability. This ablation shows that additionally using the execution trace provides more informed search direction compared with just using just the output feedback, which agrees with our hypothesis.

This experiment features the most complex graph structures, and using Trace for optimization here is similar to back-propagation over time. The experimental results here are quite impressive, showing that Trace is able to learn a complex control logic in a dozens of interactions, not only working on the training initial conditions but also on the held-out testing ones too. Nonetheless, we want to point out some limitations in the current experimental results. We find that the success rate of the learned policy varies largely across random seeds. Except for Reach (the simplest task), in a seed, often either it finds a policy close to 1.0 success rate or 0.0 success rate. Therefore, the plots can roughly be interpreted as how long it takes to find a working policy. In addition, in these experiments, we find that providing task-related context is necessary. We find the context needs to be informative enough

---

[6] The original version of OPRO uses parameter-score pairs. Since we're interested in the more general setup of OPTO, we extend it to use parameter-feedback pairs.

for humans to understand the problem[7]; otherwise, the optimization can be solved efficiently with the time scale considered here. Nonetheless, this requirement is reasonable, as there is no free lunch.

C Examples of OPTO

To ground the OPTO setup, we show how OPTO is related to some existing problems with examples.

Example 4 (Neural network with back-propagation)**.** The parameters are the weights. g is the neural computational graph and f is the loss. An example context ω can be "Minimize loss". The back-propagation algorithm, in view of the OPTO formulation, is embedded in the OPTO optimizer. For example, an OPTO optimizer here is a composition of back-propagation and gradient descent, where back-propagation takes τ to compute the propagated gradient at the parameter.

Example 5 (Code Debugging)**.** The parameters are the codes. g denotes the stacked trace and f is the error message returned by a compiler. ω can be "Make no error".

Example 6 (RL)**.** The parameters are the policy. g is the trajectory (of states, actions, rewards) resulting from running the policy in a Markov decision process; that is, g documents the graphical model of how an action generated by the policy, applied to the transition dynamics which then returns the observation and reward, etc. f can be the termination signal or a success flag. ω can be "Maximize return" or "Maximize success".

Example 7 (Hyperparameter Tuning of ML Pipeline)**.** The parameters are e.g. learning rates and architectures. g describes the stages of the ML pipeline and the evaluation on the validation set, and f is the validation loss. ω can be "Minimize validation error".

Example 8 (Prompt Optimization of an LLM Agent)**.** The parameters are the prompt of an LLM workflow. g is the computational graph of the agent and f is the feedback about the agent's behavior (which can be scores or natural language). ω can be "Maximize score" or "Follow the feedback".

Example 9 (Multi-Agent Collaboration)**.** The parameters are each agent's prompts. g describes the entire conversation flow between agents, and f is the feedback about whether the task is successful after each agent performs their action. ω can be "A group of agents coordinate to finish a task.".

As mentioned, the computational graph g returned by the Trace Oracle $\mathcal{T}$ may have different graph structures. The length of the execution trace, e.g., in the debugging example above depends on how far the code executes. Similarly, the rollout length of in the RL problem can be randomly determined. The formulation of the Trace Oracle abstracts the details of a computational workflow, so problems from different domains can be framed in the unified framework. This abstraction allows us to design the computational tool Trace for various applications.

D Trace Handles Error in Execution as Feedback

It is worth mentioning that execution error can be directly used as feedback to optimize parameters in Trace. When execution error happens within a method decorated by `bundle`, Trace would adds a special exception node to the global computational graph and throw an `TraceExecutionError` to stop the computation. The computational graph ends at where the execution error happens. This exception node becomes the new output of the inputs to the decorated method (since the original method raises an error) and is the output of the truncated computational graph. Messages in `TraceExecutionError` can then be used as the feedback f in OPTO and propagated from the exception node to the parameters. By calling an OPTO optimizer, the parameters can be updated to avoid causing the same execution error. See the exception handling code in Fig. 2.

We find that this error handling mechanism has two convenient usages. First, this allows using Trace to automatically debug issues in the workflow due to incorrect parameter settings. Such errors can happen frequently especially when codes are parameters, as during optimization codes not satisfying

---

[7] The original instructions in the v2 environments of LLF-Bench does not contain task specific background, but only the task name. We find this task name alone, except for Reach, is too vague even for humans to understand how to read the observation dictionary is related to the problem, so we added additionally one line to explain the task. Pick-place: "The goal of the task is to pick up a puck and put it to a goal position.". Push: "The goal of the task is to push a puck to a goal position.". Without this extra information, none of the LLM-based optimizers works in the experiments.

syntax or downstream API requirements can happen. The second usage is to enforce constraints the workflow has to satisfy at different stags of computation. With Trace, if an intermediate computed result does not satisfy the constraint, we can simply throw an exception which states the desired constraint. This error signal would be caught by Trace and can then provide early feedback to efficiently improve the parameters, since the graph is truncated at the error.

E Analysis of Trace

E.1 Proof of Complexity

Algorithm 2 propagates the subgraph, represented by a priority queue (implemented as a min-heap). At a time, it needs to maintain the subgraphs coming from W children separately. This leads to the space complexity of $O(WN)$. This $O(WN)$ space complexity leads to the extra $WN \log N$ factor in the time complexity of MSP compared with back-propagation, which is the time needed for merging W subgraphs of size $O(N)$.

E.2 Proof of Lower bounds

Consider an OPTO problem whose goal is to find a parameter matching a k-digit binary number. The computation checks each digit against a reference number in an arbitrary order. The feedback is either "N^{th} check failed" or "All checks succeeded". Propagated feedback must communicate k bits of information to interpret the feedback correctly; and the minimal subgraph conveys exactly that information. Updating the parameter using the minimal subgraph is trivial, whereas without it there are 2^k possibilities to check.

F When is OPTO Efficiently Solvable?

We show that OPTO covers a wide range of complicated optimization problems. This shows that if OPTO can be efficiently solved, then many complex workflows can be efficiently optimized. However, the generality of OPTO also raises some fundamental questions, such as if OPTO is well defined and when OPTO can be efficiently solved. These questions stem from its generality of the context ω and the output feedback f in OPTO, since e.g. they can be anything describable texts. This flexibility makes the scope of OPTO go beyond standard mathematical optimization problems, where a setup has a fixed context ω (e.g., "First-order optimization") and a fixed type of output feedback f (a descent direction). Fully characterize the properties of OPTO, due to its generality, is beyond the scope of this paper and would require years of future research to come. Nonetheless, here we attempt to provide some preliminary answers and point out some research questions.

F.1 What is a solution?

Classical mathematical optimization problems have a problem definition which itself is the solution concept. For example, in a minimization problem, it is clear we want to find the minimum of an objective function; even for problems as abstract and general as an equilibrium problem, the problem setup clearly states the solution concept of finding a point/set satisfying an equilibrium inequality [45]. One common pattern of these problems is that the solution concept is something that can be described as conditions on feedback that the parameter should satisfy.

By contrast, in a OPTO problem $(\Theta, \omega, \mathcal{T})$, by varying the context ω, the desired parameter can change from one extreme to another. For example ω may state "Follow the feedback" or "The feedback is adversarial.". Therefore, we need define the solution concept of OPTO differently, rather than just using the feedback. We need to also consider the context ω appropriately. Below we make an attempt to give an axiom of OPTO for its solution to be well defined.

Axiom 1 (Verifiability)**.** There is an verification oracle (a human, a machine learning model, or a polynomial-time algorithm) when given (θ, ω, f) can verify whether θ is a solution or not.

Notice the verification oracle in Axiom 1 is not limited to just algorithms. This is intentional because we currently do not have algorithms that are intelligent enough to process the wide range of contexts and feedback that OPTO allows. Therefore, we include human judgement or the use of LLMs or other AI systems as part of the definition, while acknowledging the impreciseness of the statement due to OPTO's soft computing nature. Lastly we note the verifiability is only defined with respect to the context ω and the output feedback f, not the execution trace g. That is, the verification of a solution depends only on the output of computation.

F.2 Does a solution exist?

Under Axiom 1, we can start to ask the basic question of whether a solution to an OPTO problem exists or not. There are clearly problems where no solution exists (that is, no parameter in Θ can be verified by the verification oracle). For example if the feedback f is contradicting and yet the context ω is "Follow the feedback.", then there would be no solution that is satisfactory. On the other hand, if ω is "Ignore the feedback", all parameters can be solutions. In the following, we assume solutions of OPTO under consider exist. This assumption would rule out problems, e.g., where the feedback is adversary to the context, and makes solving OPTO is a well-defined search problem.

Assumption 1. For an OPTO problem $(\theta, \omega, \mathcal{T})$, we assume there is at least a parameter $\theta \in \Theta$ such that it can be verified as a solution by the verification oracle.

F.3 Can OPTO be efficiently solved?

So far our discussion establishes OPTO as a well-defined search problem, based on qualification on the context ω and the output feedback f. However search problems can be NP-hard. In other words, we know that, without the execution trace, there are search problem instances modeled by some ω and oracle giving f that cannot be efficiently solved. Take RL for a tabular MDP as an example of OPTO problem. Without the execution trace (i.e., not seeing the Markovian structure and trajectories), the problem has an exponential complexity (due to the size of the policy space) and we know by using the execution trace here, tabular RL can be solved approximately in polynomial time [46]. Another example is training of neural networks. Without the execution trace, we have a complex black-box optimization with a loss value, without gradients, whereas an execution trace allows implementation of back-propagation to compute the gradients at the parameters.

More broadly speaking, if we consider a "human" as an optimizer for OPTO, we see that (expert) engineers/researchers, when equpped with additional computational tools, can efficiently solve a broad range of OPTO problems (such as by using the execution trace. From these observations, we conjecture using information in the execution trace is the key to unlock efficient OPTO. More precisely, we conjecture that OPTO is efficiently solvable when the context and the trace feedback need to provide information to construct a corrective search direction. For example, when the output feedback back is just a scalar loss, and yet the context + execution trace feedback does not provide enough information to compute a descending direction then OPTO reduces back to a black box problem. (See the problem instance in Appendix E.2). Nonetheless, identifying which subsets of OPTO are efficiently solvable is a big open research question.

G Additional Details of Trace and OptoPrime

G.1 Backward Step of Trace

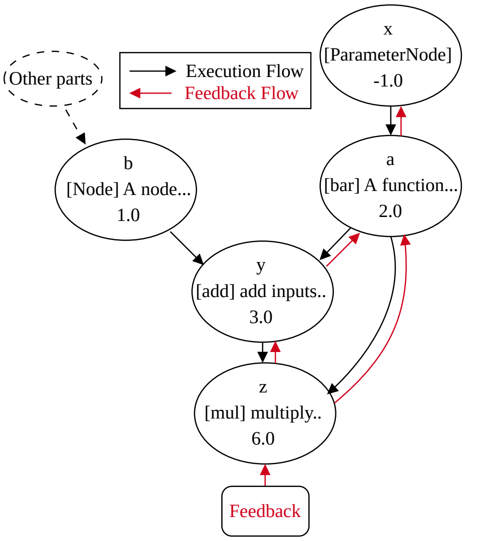
The MSP extracts the minimal subgrpah of the full computational graph of the workflow. Here we show a visualization using the example in Fig. 4.

G.2 Prompts used in OptoPrime

OptoPrime is an LLM-based optimizer. Its prompt is composed of the following parts.

1. System Prompt: Representation Prompt (Fig. A.4) + ReAct+CoT Output Prompt (Fig. A.5)
2. User Prompt (Fig. A.6 or Fig. A.7)

where + denotes concatenation. We list the prompt templates of different components below.



(a) This is an illustrative example of the graph constructed by Trace and how feedback is backpropagated to the parameter `x`.

```
#Code:
a = bar(x)
y = add(b, a)
z = mul(a, y)

#Definitions:
[mul] This is a multiply operator
[add] This is an add operator.
[bar] This is a method that does
     negative scaling.

#Inputs:
b=1.0

#Others:
a=2.0
y=3.0

#Output
z=6.0

#Variable
x=-1.0

#Feedback:
Output should be larger.
```

(b) We create a succinct summary of the computation graph using a language that mimics a program.

Figure A.3: Optimization Representation. For a program of `x = Node(-1.0); a = bar(x); y = a + 1; z = a * y` and the optimization objective of $\max_x z$, Trace automatically constructs a computation graph and represent the optimization problem as a debugging report. Note that the real program and the traced execution graph are different.

```
You're tasked to solve a coding/algorithm problem. You will see the instruction, the code, the documentation of each function used in the code, and the feedback about the execution result.

Specifically, a problem will be composed of the following parts:
- #Instruction: the instruction which describes the things you need to do or the question you should answer.
- #Code: the code defined in the problem.
- #Documentation: the documentation of each function used in #Code. The explanation might be incomplete and just contain high-level description. You can use the values in #Others to help infer how those functions work.
- #Variables: the input variables that you can change.
- #Constraints: the constraints or descriptions of the variables in #Variables.
- #Inputs: the values of other inputs to the code, which are not changeable.
- #Others: the intermediate values created through the code execution.
- #Outputs: the result of the code output.
- #Feedback: the feedback about the code's execution result.

In #Variables, #Inputs, #Outputs, and #Others, the format is:

<data_type> <variable_name> = <value>

If <type> is (code), it means <value> is the source code of a python code, which may include docstring and definitions.

```

Figure A.4: Representation Prompt that phrases the OPTO update as a pseudo-algorithm question.

```
Output_format: Your output should be in the following json format, satisfying the json syntax:

{{
"reasoning": <Your reasoning>,
"answer": <Your answer>,
"suggestion": {{
    <variable_1>: <suggested_value_1>,
    <variable_2>: <suggested_value_2>,
    }}
}}

In "reasoning", explain the problem: 1. what the #Instruction means 2. what the #Feedback on #Output means to #Variables considering how #Variables are used in #Code and other values in #Documentation, #Inputs, #Others. 3. Reasoning about the suggested changes in #Variables (if needed) and the expected result.

If #Instruction asks for an answer, write it down in "answer".

If you need to suggest a change in the values of #Variables, write down the suggested values in "suggestion". Remember you can change only the values in #Variables, not others. When <type> of a variable is (code), you should write the new definition in the format of python code without syntax errors, and you should not change the function name or the function signature.

If no changes or answer are needed, just output TERMINATE.

```

Figure A.5: ReAct+CoT Output Prompt that instructs LLMs should respond in the format of (reasoning, answer, suggestion) and explains the output format.

```
Now you see problem instance:

================================
{actual_problem_instance}
================================

Your response:

```

Figure A.6: User Prompt for OptoPrime without Memory

```
Now you see problem instance:

================================
{actual_problem_instance}
================================

Below are some variables and their feedbacks you received in the past.

{
    "variables": {
        {variable1_name}: {variable1_value1}
        {variable2_name}: {variable2_value1}
        ...
    },
    "feedback": {feedback_1}
}

{
    "variables": {
        {variable1_name}: {variable1_value2}
        {variable2_name}: {variable2_value2}
        ...
    },
    "feedback": {feedback_2}
}
...


Your response:

```

Figure A.7: User Prompt for OptoPrime with Memory

```
#Instruction
{instruction}

#Code
{code}

#Documentation
{documentation}

#Variables
{variables}

#Constraints
{constraints}

#Inputs
{inputs}

#Others
{others}

#Outputs
{outputs}

#Feedback:
{feedback}
```

Figure A.8: Problem Template used to fill the User Prompt. By default the Instruction (which is the context ω of OPTO) is "You need to change the <value> of the variables in #Variables to improve the output in accordance to #Feedback."

H Examples of the Optimized Parameters in the Experiments

```
## Iteration 0 ( initialization ) ; Success : False
def reason(self , map) -> str:
    """
    Given a map, analyze the board in a game. On map, O denotes misses, X denotes successes, and . denotes unknown positions.
    """
    # Check for first unknown position to try as a potential move
    for row_index , row in enumerate(map):
        for col_index , val in enumerate(row):
            if val == ’.’:
                return [row_index , col_index]
    # Default to [0, 0] if no unknown positions found
    return [0, 0]

## Iteration 2 ; Success : True
def reason(self , map) -> str:
    """
    Given a map, analyze the board in a game. On map, O denotes misses, X denotes successes, and . denotes unknown positions.
    """
    # Improved strategic approach by checking . surrounded by more . which might indicate undetected area
    max_surrounding = -1
    best_spot = [0, 0]
    for row_index , row in enumerate(map):
        for col_index , val in enumerate(row):
            if val == ’.’:
                surrounding = 0
                for di in [-1, 0, 1]:
                    for dj in [-1, 0, 1]:
                        ni , nj = row_index + di, col_index + dj
                        if 0 <= ni < len(map) and 0 <= nj < len(map[0]) and map[ni][nj] == ’.’:
                            surrounding += 1
                if surrounding > max_surrounding:
                    max_surrounding = surrounding
                    best_spot = [row_index , col_index]
    return best_spot

## Iteration 8 ; Success : True
def reason(self , map) -> str:
    """
    Given a map, analyze the board in a game. On map, O denotes misses, X denotes successes, and . denotes unknown positions.
    """
    max_surrounding = -1
    best_spot = [0, 0]
    for row_index , row in enumerate(map):
        for col_index , val in enumerate(row):
            if val == ’.’:
                surrounding = 0
                for di in [-1, 0, 1]:
                    for dj in [-1, 0, 1]:
                        ni , nj = row_index + di, col_index + dj
                        if 0 <= ni < len(map) and 0 <= nj < len(map[0]):
                            if map[ni][nj] == ’.’:
                                surrounding += 1
                            elif map[ni][nj] == ’X’:
                                surrounding += 1.5  # increasing emphasis on proximity to successful hits
                if surrounding > max_surrounding:
                    max_surrounding = surrounding
                    best_spot = [row_index , col_index]
    return best_spot

```

Figure A.9: Learned Battleship Code. Functions with the same name are learned during different iterations or trials.

```python
## Iteration 0 ( initialization )
def create_prompt(self, prompt_template, question):
    """
    The function takes in a question and then add to the prompt for LLM to answer.
    Args:
        prompt_template: some guidance/hints/suggestions for LLM
        question: the question for the LLM to answer
    """
    return prompt_template.format(question)

## Iteration > 0
def create_prompt(self, prompt_template, question):
    """
    The function takes in a question and then add to the prompt for LLM to answer.
    The prompt should now further instruct the LLM to carefully track the ball swaps occurring step-by-step.
    Args:
        prompt_template: some guidance/hints/suggestions for LLM
        question: the question for the LLM to answer
    """
    prompt_template = 'Process this carefully: Step-by-step.' + prompt_template
    return prompt_template.format(question)

```

Figure A.10: Learned BigBench Code. Functions with the same name are learned during different iterations or trials.

```
## Iteration 0 ( initialization )
def extract_answer(self, prompt_template, question, response):
    """
    Need to read in the response, which can contain additional thought, delibration and an answer.
    Use code to process the response and find where the answer is.
    Can use self.call_llm("Return the answer from this text: ” + response) again to refine the answer if necessary.

    Args:
        prompt_template: The prompt that was used to query LLM to get the response
        question: Question has a text describing the question but also "Options”
        response: LLM returned a string response
                  Process it and return the answer in the exact format that the evaluator wants to see.
                  Be mindful of the type of answer you need to produce.
                  It can be (A)/(B), a number like 8, or a string, or Yes/No.
    """
    answer = response.split("Answer:")[1].strip()
    return answer

## Iteration > 0
def extract_answer(self, response):
    """
    Need to read in the response, which can contain additional thought, deliberation and an answer.
    Use code to process the response and find where the answer is.
    Can use self.call_llm("Return the answer from this text: ” + response) again to refine the answer if necessary.
    Args:
        response: LLM returned a string response
                  Process it and return the answer in the exact format that the evaluator wants to see.
                  Be mindful of the type of answer you need to produce.
                  It can be (A)/(B), a number like 8, or a string, or Yes/No.
        question: Question has a text describing the question but also "Options”
    """
    answer = ’’
    segments = response.split(’\n’)
    for segment in segments:
        if ’Answer:’ in segment:
            answer = segment.split(’Answer:’)[1].strip()
    refined_answer = self.call_llm(’Return the refined answer from this text: ’ + answer)
    return refined_answer

def extract_answer(self, prompt_template, question, response):
    """
    Processes the LLM response and extracts the final answer in the required format.
    """
    # Assuming that the relevant part of the response is after ’Answer:’ and before any further commentary
    extracted_part = response.split(’Answer: ’)[1].split(’ ’)[0].strip()
    # Find the section of the answer and return it directly
    result = re.search(’\([A-E]\)’, extracted_part)
    if result:
        return result.group()
    else:
        return ’No valid answer found’

def extract_answer(self, prompt_template, question, response):
    """
    Processes the LLM response, extracting and formatting the final answer.
    Uses code to meticulously parse the response to locate the answer section.
    Optionally refines the answer by querying the LLM again if necessary.

    Args:
        response: string from LLM, expected format contains ’Answer:’ followed by the answer.
        question: Description of the question being addressed, may include ’Options’
    """
    answer = response.split(’Answer:’)[1].split(’.’)[0].strip().lower().split(’,’)[0].strip()
    return answer)

```

Figure A.11: Learned BigBench Code. Functions with the same name are learned during different iterations or trials.

```
## Iteration 0 ( initialization )
prompt_template = """
Given the fields ‘question‘, produce the fields ‘answer‘.

---

Follow the following format.

Question:
Answer:

---
Question: {}
Answer:
"""

## Iteration >0
prompt_template = """
Given the fields ‘question‘, produce the fields ‘answer‘.

---

Follow the following format.

Question:
Answer: Please respond precisely in the format required.

---
Question: {}
"""
```

Figure A.12: Learned BigBench Prompt Template.

```

## Iteration 0 (initialization); Success: False
def controller(obs):
    """
    A feedback controller that computes the action based on the observation.

    Args:
        obs: (dict) The observation from the environment. Each key is a string (indicating a type of observation) and the value is a list of floats.
    Output:
        action: (list or nd.array) A 4-dimensional vector.
    """
    return [0, 0, 0, 0]

## Iteration 1; Success: False
def controller(obs):
    """
    A feedback controller that computes the action based on the observation.

    Args:
        obs: (dict) The observation from the environment. Each key is a string (indicating a type of observation) and the value is a list of floats.
    Output:
        action: (list or nd.array) A 4-dimensional vector.
    """
    hand_pos = obs[’hand_pos’]
    goal_pos = obs[’goal_pos’]
    action = [goal_pos[0] - hand_pos[0], goal_pos[1] - hand_pos[1], goal_pos[2] - hand_pos[2], 0]
    # Adjusting the last element of the action based on whether the hand is close enough to trigger grasping
    if abs(goal_pos[0] - hand_pos[0]) < 0.02 and abs(goal_pos[1] - hand_pos[1]) < 0.02 and abs(goal_pos[2] - hand_pos[2]) < 0.02:
        action[3] = 1 # Assuming action[3] = 1 triggers closing the gripper
    return action

## Interation 5; Success: False
def controller(obs):
    """
    A feedback controller that computes the action based on the observation.

    Args:
        obs: (dict) The observation from the environment.
    Output:
        action: (list or nd.array) A 4-dimensional vector.
    """
    hand_pos = obs[’hand_pos’]
    puck_pos = obs[’puck_pos’]
    goal_pos = obs[’goal_pos’]
    gripper_open = 1 if obs[’gripper_distance_apart’][0] > 0.5 else 0

    # Adjusting strategy to close the gripper when in proximity of the puck
    if gripper_open and ((abs(hand_pos[0] - puck_pos[0]) < 0.05) and (abs(hand_pos[1] - puck_pos[1]) < 0.05) and (abs(hand_pos[2] - puck_pos[2]) < 0.05)):
        action = [0, 0, 0, 1]  # Close the gripper
    elif not gripper_open:
        direction_to_goal = [0.09 - hand_pos[0], 0.95 - hand_pos[1], 0.12 - hand_pos[2]]
        action = [direction_to_goal[0], direction_to_goal[1], direction_to_goal[2], 0]  # Move towards the suggested pose once puck is grasped
    else:
        # Move towards the puck first if not carrying it
        direction_to_puck = [puck_pos[0] - hand_pos[0], puck_pos[1] - hand_pos[1], puck_pos[2] - hand_pos[2]]
        action = [direction_to_puck[0], direction_to_puck[1], direction_to_puck[2], 0]

    return action
```

Figure A.13: Learned Code for LLFBench Meta-World Pick-Place (Part 1). Functions with the same name are learned during different iterations or trials.

```


## Iteration 10; Success: False
def controller(obs):
    """
    A feedback controller that computes the action based on the observation.

    Args:
        obs: (dict) The observation from the environment.
    Output:
        action: (list or nd.array) A 4-dimensional vector.
    """
    hand_pos = obs['hand_pos']
    puck_pos = obs['puck_pos']
    goal_pos = obs['goal_pos']
    gripper_open = 1 if obs['gripper_distance_apart'][0] > 0.5 else 0

    normalize = lambda x: [i / max(abs(max(x, key=abs)), 1) for i in x]

    # Close the gripper when close to the puck and the gripper is open
    if gripper_open and ((abs(hand_pos[0] - puck_pos[0]) < 0.05) and (abs(hand_pos[1] - puck_pos[1]) < 0.05) and (abs(hand_pos[2] - puck_pos[2]) < 0.05)):
        return [0, 0, 0, 1]
    # When the puck is grasped, move towards the goal position with a normalized direction
    elif not gripper_open and ((abs(hand_pos[0] - puck_pos[0]) < 0.1) and (abs(hand_pos[1] - puck_pos[1]) < 0.1) and (abs(hand_pos[2] - puck_pos[2]) < 0.1)):
        direction_to_goal = [goal_pos[0] - hand_pos[0], goal_pos[1] - hand_pos[1], goal_pos[2] - hand_pos[2]]
        return normalize(direction_to_goal) + [0]
    # When the gripper is open and not close enough to the puck, move towards the puck
    else:
        direction_to_puck = [puck_pos[0] - hand_pos[0], puck_pos[1] - hand_pos[1], puck_pos[2] - hand_pos[2]]
        return normalize(direction_to_puck) + [0]

## Iteration 13; Success: True
def controller(obs):
    """
    A feedback controller that computes the action based on the observation.

    Args:
        obs: (dict) The observation from the environment.
    Output:
        action: (list or nd.array) A 4-dimensional vector.
    """
    hand_pos = obs['hand_pos']
    puck_pos = obs['puck_pos']
    goal_pos = obs['goal_pos']
    gripper_open = 1 if obs['gripper_distance_apart'][0] > 0.5 else 0

    normalize = lambda x: [i / max(abs(max(x, key=abs)), 1) for i in x]
    # Close the gripper when close to the puck and the gripper is open
    if gripper_open and ((abs(hand_pos[0] - puck_pos[0]) < 0.05) and (abs(hand_pos[1] - puck_pos[1]) < 0.05) and (abs(hand_pos[2] - puck_pos[2]) < 0.05)):
        action = [0, 0, 0, 1]  # Close the gripper
    elif not gripper_open and ((abs(hand_pos[0] - puck_pos[0]) < 0.1) and (abs(hand_pos[1] - puck_pos[1]) < 0.1) and (abs(hand_pos[2] - puck_pos[2]) < 0.1)):
        # When the puck is grasped, adjust direction towards the goal with improved precision and ensure successful pickup
        direction_to_goal = normalize([goal_pos[0] - hand_pos[0], goal_pos[1] - hand_pos[1], goal_pos[2] - hand_pos[2]])
        action = direction_to_goal + [1]  # Keep the gripper closed
    else:
        # Move towards the puck first if not carrying it
        direction_to_puck = normalize([puck_pos[0] - hand_pos[0], puck_pos[1] - hand_pos[1], puck_pos[2] - hand_pos[2]])
        action = direction_to_puck + [0]  # Open the gripper to prepare for grasping

    return action
```

Figure A.14: Learned Code for LLFBench Meta-World Pick-Place (Part 2). Functions with the same name are learned during different iterations or trials.

```
 # Iteration 1
 {
    "reasoning": "The task involves controlling a robotic arm to perform a pick-place
     operation by interpreting observations and generating appropriate control actions (4D
     vectors) to move the arm towards the goal. The objective is to adjust the controller's
     code (__code0) based on feedback received at each step of the process. The feedback within
     #Others suggests whether the movements made by the arm are leading it closer to or
     farther from the goal, along with advice on what coordinates to aim for next (e.g.,
     [-0.07  0.68  0.12  0.  ]). The controller's code currently returns a static action [0,
     0, 0, 0], which does not adjust based on observations and hence does not efficiently
     move the robotic arm towards the goal. Given the repetitive feedback suggesting to go to
     [-0.07  0.68  0.12  0.  ], it appears that the controller needs to dynamically compute
     actions based on the difference between the current positions (hand_pos, puck_pos) and
     the goal_pos, along with possibly manipulating the gripper state for pick and place
     operation.",
    "answer": "",
    "suggestion": {...}
}


# Iteration 5
 {
    "reasoning": "The instruction involves creating a controller for a Sawyer robot arm to
     solve a pick-and-place task by providing sequential control actions based on
     observations of the environment and state of the robot. The feedback indicates that
     despite the actions taken, the success criteria have not been met, with the latest
     attempt achieving a return of 0.9574427394250022. The code provided for __code0 defines
     a controller function using the observed state to compute control actions aimed at
     grasping a puck and moving it to a goal position while adjusting the gripper based on
     its distance from the puck. Feedback throughout the execution indicates that the
     controller adapts its strategy based on proximity to the puck and whether it has been
     picked up. Feedback also suggests that the gripper's opening and closing actions are
     central to the task's success, with corrections being made to close the gripper when
     necessary. The controller's logic for determining movements toward the puck or goal and
     adjusting the gripper state appears reasonable, but the feedback implies that
     adjustments might be required to ensure task completion. This could include fine-tuning
     the conditions under which the gripper is closed or adjusting the thresholds for puck
     pickup and delivery to the goal.",
    "answer": "",
    "suggestion": {}

# Iteration 10
 {
    "reasoning": "The task involves programming a robot arm to pick up a puck and place it
     at a goal position by providing sequential 4-dimensional action vectors. The action
     vectors represent target positions and gripper states (open or close) based on the
     robot's and puck's current positions. The feedback indicates that despite optimizing the
     controller logic to close the gripper when near the puck, move towards the puck when
     it's not grasped, and head towards the goal once the puck is grasped, the actions have
     not resulted in task completion. Specifically, repeated suggestions to close the
     gripper (indicating it remained open when it should have been closed) and guidance on
     where to move next suggest that the action selection logic might not be effectively
     responding to the environment's state or might not be properly engaging the gripper at
     the right times. The latest code includes a condition to normalize the direction of
     movement and checks for gripper position, but there seems to be a misunderstanding or
     oversight in adjusting the robot's behavior based on the scenario to successfully
     complete the pick-place task.",
    "answer": "",
    "suggestion": {...}
}
}
```

Figure A.15: Example Responses of LLM for LLFBench Meta-World Pick-Place (Part 1); Content in suggestion is omitted for space (represented as ...); please see the code examples.

```
# Iteration 12
 {
    ”reasoning”: ”In this task, the controller function for the Sawyer robot arm manipulates
     the robot’s actions based on the observed environment to achieve a pick-and-place task.
     The function correctly defines actions based on the robot’s hand position, the puck’s
     position, the goal position, and the gripper’s state. However, the feedback after each
     step in the documentation indicates that although the robot is making progress towards
     the goal, there is an issue with the gripper’s action state not being switched to
     ’close’ (1) at the crucial moment when the robot is close enough to the puck. Most
     feedback comments suggest that the robot is moving in the right direction but fails to
     grasp the puck due to the gripper not closing. This inefficiency appears to arise from
     the conditions set for closing the gripper and possibly the normalized direction
     towards the goal after the puck has been grasped. The condition for closing the gripper
     (when close enough to the puck) seems to be met, but the action selected afterwards
     does not effectively lead to puck manipulation or goal achievement in the most
     optimized manner, as indicated by the ’Feedback’ messages suggesting movements that
     imply needing adjustments in action selection.”,
    ”answer”: ””,
    ”suggestion”: {...}
}

# Iteration 13 (Success)
LLM response:
 {
    ”reasoning”: ”The instruction requests improvements based on feedback for a Sawyer robot
     arm’s controller function, aiming to accomplish a pick-and-place task. The controller
     function accepts observations about the robot and the environment and computes an
     action to move the robot accordingly. \n\nThe feedback provided indicates the task was
     successfully completed with a suggestion that the final command should have been
     ‘[0.04, 0.88, 0.24, 1]‘, which probably indicates the precise position and gripper
     state (closed) needed to achieve the task goal. The feedback suggests the need for
     fine-tuning actions especially towards the end of the task to ensure the robot’s
     gripper is appropriately positioned and the gripper state is correct for completing the
     task. Since the feedback for the current code positively notes ’Success: True’ with a
     high return value, no modifications are recommended for ‘__code0‘. \n\nPrevious feedback
     instances with lower success metrics and recommendations for change highlight the
     iterative improvements to the controller logic, specifically related to grip control
     and movement precision. The aim was to adjust the controller’s response to observation
     inputs to accurately position the gripper and handle the puck. The current code
     evidently manages these aspects well by normalizing directions and managing the
     gripper’s open-close state effectively under varying circumstances (proximity to puck
     or goal and whether the puck is already grasped). These adjustments have likely led to
     the successful output in the described problem.”,
    ”answer”: ”No changes are recommended as the feedback indicates successful task
     completion and suggests the controller’s current logic adequately addresses the task’s
     requirements.”,
    ”suggestion”: {}
}
```

Figure A.16: Example Responses of LLMs for LLFBench Meta-World Pick-Place (Part 2); Content in suggestion is omitted for space (represented as ...); please see the code examples.

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: The contributions (Trace, OPTO, OptoPrime) are clearly stated in the introduction and detailed in Sections 2,3,4, and claimed improvements are validated in Section 5. Meanwhile the aspirational goal of self-improving interactive agents is elaborated separately in the concluding Section 7.

   Guidelines:

   - The answer NA means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A No or NA answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: The limitations are discussed in Section 6.

   Guidelines:

   - The answer NA means that the paper has no limitation while the answer No means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory Assumptions and Proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

Answer: [Yes]

Justification: There are theorems about the computational complexity of Trace and an information theoretic lower bound on propagated feedback. Both theorems are proved in Appendix E.

Guidelines:

- The answer NA means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental Result Reproducibility**

Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

Answer: [Yes]

Justification: Experimental setups are outlined in Section 5 and detailed in Appendix B. Supplementary material additionally provides the exact code that was run to produce all results, and only requires user to supply an OpenAI API key.

Guidelines:

- The answer NA means that the paper does not include experiments.
- If the paper includes experiments, a No answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
- If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
- Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
- While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
    (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
    (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
    (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
    (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

   Answer: [Yes]

   Justification: All of the code to implement Trace and OptoPrime are provided in the supplementary material. All of the experiments described in the paper are reproducible with the supplied code.

   Guidelines:

   - The answer NA means that paper does not include experiments requiring code.
   - Please see the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - While we encourage the release of code and data, we understand that this might not be possible, so “No” is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
   - The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://nips.cc/public/guides/CodeSubmissionPolicy`) for more details.
   - The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
   - The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
   - At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
   - Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental Setting/Details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer, etc.) necessary to understand the results?

   Answer: [Yes]

   Justification: Experimental setups are outlined in Section 5 and detailed in Appendix B.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment Statistical Significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [Yes]

   Justification: All experiments include standard error bars from ≥ 10 replications.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The authors should answer "Yes" if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.

- The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
- The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
- The assumptions made should be given (e.g., Normally distributed errors).
- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g. negative error rates).
- If error bars are reported in tables or plots, The authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments Compute Resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: All the experiments use the same compute resources (LLM API for OptoPrime and machine to run Trace), which are listed at the start of Section 5.

   Guidelines:

   - The answer NA means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code Of Ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: The contributions of this paper (a framework analogous to PyTorch to generalize the backpropagation algorithm) do not have direct safety or security implications.

   Guidelines:

   - The answer NA means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer No, they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader Impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [Yes]

    Justification: In Sections 6 and 7 we discuss both the aspirational goals and their broader impacts.

    Guidelines:

- The answer NA means that there is no societal impact of the work performed.
- If the authors answer NA or No, they should explain why their work has no societal impact or why the paper does not address societal impact.
- Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.
- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pretrained language models, image generators, or scraped datasets)?

Answer: [NA]

Justification: There are no models, data or APIs released along with the paper.

Guidelines:

- The answer NA means that the paper poses no such risks.
- Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
- Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
- We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

Answer: [Yes]

Justification: Datasets used in the experiments (e.g. BigBench-Hard) are properly attributed.

Guidelines:

- The answer NA means that the paper does not use existing assets.
- The authors should cite the original paper that produced the code package or dataset.
- The authors should state which version of the asset is used and, if possible, include a URL.
- The name of the license (e.g., CC-BY 4.0) should be included for each asset.
- For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.

- If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.
- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New Assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [Yes]

    Justification: There is code implementing Trace and OptoPrime that are released along with the paper. The code repository contains licenses and several tutorial notebooks documenting each functionality. There are no datasets or models in the release, hence datasheets and modelsheets are not applicable for this release.

    Guidelines:

    - The answer NA means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and Research with Human Subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [NA]

    Justification: There are no crowdsourcing or human subject studies conducted in this paper.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional Review Board (IRB) Approvals or Equivalent for Research with Human Subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [NA]

    Justification: The paper does not involve crowdsourcing nor research with human subjects.

    Guidelines:

    - The answer NA means that the paper does not involve crowdsourcing nor research with human subjects.

- Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.
- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.