
Supplementary information

Optimizing generative AI by backpropagating language model feedback

In the format provided by the
authors and unedited

Supplementary Note 1 TextGrad details

Supplementary Note 1.1 Variables

Variables are the nodes in the computation graph. Below are the most important attributes of **Variables**:

1. **Value** is the unstructured data that the variable contains. Throughout this manuscript, all values are text data.
2. **Role description** is an informative string that describes the role of the variable in the computation graph. We use these roles to let the user inject knowledge into the graph and guide the optimization behavior. More information is described below.
3. **Gradients** are the natural language feedback provided by the LLMs during the backward pass. These describe the changes to make the variable so that the downstream loss can be improved. For an example backward operation that populates gradients, please read [Supplementary Note 1.3](#).
4. **Predecessors** are the set of variables that are used to generate a given variable. For instance, if we are giving an instruction to an LLM and getting a response, the instruction would be the predecessor of the response. During the backward pass, the gradients on the successor are passed to its predecessors, to provide guidance around how to improve the downstream objective.
5. **Requires grad** indicates whether or not the gradients will be populated during the backward pass, analogous to PyTorch. For instance, if the user does not wish to compute gradients for a question, then simply write `Variable(value=question, requires_grad=False, ...)` to indicate this.

Role description. In TextGrad, each variable has a *role description*. In particular, this is a string that describes the role of the variable in the computation graph, such as “system prompt to the language model” or “prediction by the language model”. While sometimes populated automatically, in general role descriptions are one of the primary ways to inject user knowledge into the optimization process.

Empirically, we find that role descriptions can significantly steer the optimization process. For instance, setting the role of a prediction to be “the final numerical answer to the language model” guides the Textual Gradient Descent optimizer, that prompts a language model to update the value of the variable using the feedback, to update the variable such that it is only a numerical value. In comparison, a role description such as “the reasoning for the solution and the final prediction” guides the optimizer to produce the reasoning along with the final prediction.

Here is an example usage:

```

1   import textgrad as tg
2
3   system_prompt = tg.Variable("You will be given a question and think step
    -by-step.", requires_grad=True, role_description="system prompt to the
    language model that will be reused across queries")
4
5   model = tg.BlackBoxLLM(system_prompt=system_prompt)

```

Code Snippet 1 An example usage of a role description.

Supplementary Note 1.2 Backpropagation

The backpropagation algorithm in TextGrad mirrors existing autograd frameworks for deep learning. See Algorithm 1 for the pseudocode.

Algorithm 1 Backpropagation in TextGrad

```

1: Input: Variables  $v \in \mathcal{V}$  in a graph, loss variable  $\mathcal{L}$ , backward engine (LLM)  $\mathcal{M}$ 
   that will provide textual gradients
2: # Initializing gradients
3: for each  $v \in \mathcal{V}$  do
4:    $v.\text{gradients} = \emptyset$ 
5: end for
6: # Topological Sorting
7:  $Q \leftarrow \text{TopologicalSort}(G)$ 
8: # Backpropagation
9: for  $v$  in  $Q$  do
10:  # Populate gradients in predecessors
11:  for each  $u \in \text{PredecessorsOf}(v)$  do
12:    # We omit subscript  $v$  in  $f$ .  $f$  is the function that generates
     $v$ , and  $\nabla_{\mathbf{f}}$  is the backward operation for that function.
13:    # Semantically, it provides feedback to the variable  $u$ , given
    how  $v$  is produced, and the feedback we already collected for  $v$ .
14:     $u.\text{gradients.add}(\nabla_{\mathbf{f}}(u, v, \frac{\partial \mathcal{L}}{\partial v}))$ 
15:  end for
16: end for

```

Supplementary Note 1.3 Functions

TextGrad offers several operations where both the forward and backward computations are defined – as such, these operations are composable. The abstract `textgrad.autograd.Function` class has two methods: `forward` and `backward`, mirroring the PyTorch syntax. Each function has to define both of these methods. Below, we describe a couple of the most used operations in this paper.

LLMcall function. Currently, the most crucial operation in TextGrad is the call to language models.

Forward mode. The forward mode is simple: We make a call to an LLM, through an API or through the local machine. When a call is made, all the input variables are registered as the predecessors of the response from the LLM. For instance, if we ask a question to an LLM using an `instruction` and a `question` variable, the `response` variable's predecessors will be `[instruction, question]`. When doing the backward pass, the gradients on the `response` will be backpropagated to `question` and `instruction`.

Backward mode. To ensure the backward function runs smoothly and generally, we add the following glossary to the system prompt:

Glossary for backward mode of the LLMcall function

```
### Glossary of tags that will be sent to you:
# - <LM_SYSTEM_PROMPT>: The system prompt for the language
model.
# - <LM_INPUT>: The input to the language model.
# - <LM_OUTPUT>: The output of the language model.
# - <OBJECTIVE_FUNCTION>: The objective of the optimization task.
# - <VARIABLE>: Specifies the span of the variable.
# - <ROLE>: The role description of the variable.
```

Using these tags, the LLM is made aware of the objective, the role and the value of the variable to give feedback to, and the full conversation in the forward mode.

This glossary is provided in the system prompt to the backward engine LLM:

System prompt for the backward mode of the LLMcall function

You are part of an optimization system that improves a given text (i.e. the variable). You are the gradient (feedback) engine. Your only responsibility is to give intelligent and creative feedback and constructive criticism to variables, given an objective specified in `<OBJECTIVE_FUNCTION>` and `</OBJECTIVE_FUNCTION>` tags. The variables may be solutions to problems, prompts to language models, code, or any other text-based variable. Pay attention to the role description of the variable, and the context in which it is used. You should assume that the variable will be used in a similar context in the future. Only provide strategies, explanations, and methods to change in the variable. DO NOT propose a new version of the variable, that will be the job of the optimizer. Your only job is to send feedback and criticism (compute ‘gradients’). For instance, feedback can be in the form of ‘Since language models have the X failure mode...’, ‘Adding X can fix this error because...’, ‘Removing X can improve the objective function because...’, ‘Changing X to Y would fix the mistake ...’, that gets at the downstream objective.

If a variable is already working well (e.g. the objective function is perfect, an evaluation shows the response is accurate), you should not give feedback.
{GLOSSARY}

Most of this setup aims to ensure that the user does not have to modify the gradient computation. All of our experiments in the diverse set of applications are done with the same backward mode.

For instance, an example prompt for the gradient computation looks like the following:

An example backward mode prompt for the LLMCall function

You will give feedback to a variable with the following role: <ROLE> system prompt to a language model </ROLE>.

Here is a conversation with a language model (LM):

<LM_SYSTEM_PROMPT> You will answer a reasoning question. Think step by step. Always conclude the last line of your response should be of the following format: ‘Answer: \$VALUE’ where VALUE is a numerical value. </LM_SYSTEM_PROMPT>

<LM_INPUT> I have a chicken, a frog, a mouse, a cat, two pigs, and two rabbits. How many animals do I have? </LM_INPUT>

<LM_OUTPUT> To find the total number of animals, we simply add up the number of each type of animal: 1 chicken + 1 frog + 1 mouse + 1 cat + 2 pigs + 2 rabbits = 8 animals.
Answer: 8 </LM_OUTPUT>

This conversation is part of a larger system. The <LM_OUTPUT> was later used as response from the language model.

<OBJECTIVE_FUNCTION> Your goal is to give feedback to the variable to address the following feedback on the <LM_OUTPUT>. To improve the runtime of the string-based function that checks if the prediction is correct, consider the following feedback:

1. ****Simplify the Response****: For example, instead of “To find the total number of animals, we simply add (...) + 2 pigs + 2 rabbits = 8 animals. Answer: 8”, a more concise response like “Total animals: 8” would be more efficient.

...

By implementing these strategies, the response from the language model can be optimized to improve the runtime of the string-based function that checks if the prediction is correct. </OBJECTIVE_FUNCTION>

We are interested in giving feedback to the system prompt to a language model. Specifically, give feedback to the following span of text:

<VARIABLE> You will answer a reasoning question. Think step by step. (...) following format: ‘Answer: \$VALUE’ where VALUE is a numerical value. </VARIABLE>

Given the above history, describe how the system prompt to a language model could be improved to improve the <OBJECTIVE_FUNCTION>. Be very creative, critical, and intelligent.

Addition operation. In numerical optimization, we have the following:

$$z = x + y, \tag{A1}$$

$$w = t + z, \tag{A2}$$

$$\frac{\partial w}{\partial x} = \frac{\partial w}{\partial z} \frac{\partial z}{\partial x} = \frac{\partial w}{\partial z}. \tag{A3}$$

In numerical derivatives, due to the linearity of addition, we have:

$$\frac{\partial \sum_i^N L_i}{\partial x} = \sum_i^N \frac{\partial L_i}{\partial x}. \tag{A4}$$

In particular, the backward function for the addition operation passes the gradients on the output of the addition operation to its inputs:

$$\text{tg.sum}(L_1, \dots, L_N) = “\{L_1\} \setminus n\{L_2\} \setminus n \dots \{L_N\}” . \tag{A5}$$

Similarly, in TextGrad, we have the `tg.sum` operation, which lets the gradients (feedback) on the output variable pass to the input variables:

$$\frac{\partial \text{tg.sum}(L_1, \dots, L_N)}{\partial x} = \text{tg.sum}\left(\frac{\partial L_1}{\partial x}, \dots, \frac{\partial L_N}{\partial x}\right) = \left\{ \frac{\partial L_1}{\partial x} \right\} \backslash n \dots \backslash n \left\{ \frac{\partial L_N}{\partial x} \right\}. \quad (\text{A6})$$

where we use $\backslash n$ to indicate concatenation.

Use case. One canonical use case for the addition operation is the batch optimization case. In particular, we implement minibatch gradient descent when performing prompt optimization (3).

Supplementary Note 1.4 Textual gradient descent implementation

Similar to backward computations, we strive to preserve generality in the implementation of TGD. In particular, we use the same glossary of tags provided above to inject information to the optimization process.

The current system prompt to the optimizer call is the following:

System prompt for the TextualGradientDescent optimizer.

You are part of an optimization system that improves text (i.e., variable). You will be asked to creatively and critically improve prompts, solutions to problems, code, or any other text-based variable. You will receive some feedback, and use the feedback to improve the variable. The feedback may be noisy, identify what is important and what is correct. Pay attention to the role description of the variable, and the context in which it is used. This is very important: You MUST give your response by sending the improved variable between `<IMPROVED_VARIABLE>` improved variable `</IMPROVED_VARIABLE>` tags. The text you send between the tags will directly replace the variable.
{GLOSSARY}

Below is an example prompt to the optimizer:

An example prompt to the optimizer

Here is the role of the variable you will improve: <ROLE> system prompt to the language model </ROLE>.

The variable is the text within the following span: <VARIABLE> You will answer a reasoning question. Think step by step. (...) following format: 'Answer: \$VALUE' where VALUE is a numerical value. </VARIABLE>

Here is the context and feedback we got for the variable:
<CONTEXT>

Here is a conversation:
<CONVERSATION>
<LM_SYSTEM_PROMPT> You will answer a reasoning question. Think step by step. The last line of your response should be of the following format: 'Answer: \$VALUE' where VALUE is a numerical value. </LM_SYSTEM_PROMPT>
<LM_INPUT> I have a carrot, an onion, a stalk of celery, three yams, a garlic, a head of broccoli, and a potato. How many vegetables do I have? </LM_INPUT>
<LM_OUTPUT> To determine the number of vegetables you have, we need to count each individual vegetable item.
1. Carrot 2. Onion 3. Celery 4. Yams (3 yams) 5. Garlic 6. Broccoli 7. Potato
Counting all the individual vegetable items, we have a total of 7 vegetables.
Therefore, you have 7 vegetables.
Answer: 7 </LM_OUTPUT>
</CONVERSATION>

This conversation is potentially part of a larger system. The output is used as response from the language model.

Here is the feedback we got for system prompt to the language model in the conversation:
<FEEDBACK>
To improve the structured system prompt for the language model, consider the following feedback:
1. **Clarify the Calculation Process**:
- **Current Issue**: The prompt does not explicitly instruct the model to outline each step of its calculation process.
- **Improvement**: Add a directive that requires the model to explicitly state each quantity and the sum. For example, "Clearly outline each step of your calculation process, stating each quantity and the sum."
2. **Verify Intermediate Steps**:
- **Current Issue**: The prompt does not instruct the model to verify each intermediate step.
- **Improvement**: Include a directive for the model to verify each step against known correct values. For example, "Verify each intermediate step in your calculation to ensure accuracy."
</FEEDBACK>

</CONTEXT>

Improve the variable (system prompt to the language model) using the feedback provided in <FEEDBACK> tags.

Send the improved variable in the following format: <IMPROVED_VARIABLE> the improved variable </IMPROVED_VARIABLE>

Send ONLY the improved variable between the <IMPROVED_VARIABLE> tags, and nothing else.

Supplementary Note 2 Optimizer extensions

Supplementary Note 2.1 Batch optimization

In batch optimization, we use the `tg.sum` function described above. In particular, gradients propagating from multiple instances are concatenated together, thus the optimizer sees all of the feedback to a variable coming from multiple sources.

The syntax is as simple as the following:

```
1 losses = [loss_fn(answer, model(question)) for question, answer in batch]
2 total_loss = tg.sum(losses)
3 total_loss.backward()
```

Code Snippet 2 An example use for batch optimization for question answering.

Supplementary Note 2.2 Constrained optimization with natural language constraints

In TextGrad it is possible use constraints when optimizing variables. These constraints are all defined as natural language descriptions. For example, one can prompt optimizer to update the variable but to conclude its response with an answer during the update:

Example constrained optimization prompt

You must follow the following constraints:

<CONSTRAINTS> *"The last line of your response should be of the following format: 'Answer: \$LETTER' where LETTER is one of ABCD."*
</CONSTRAINTS>

In general, the constraint post-fix is appended to the optimizer's prompt, where the constraints are written within the <CONSTRAINTS> {constraint text} </CONSTRAINTS> tags.

In code, the user can simply pass in the constraints to the TGD optimizer:

```
1 constraints = ["The last line of your response should be of the following
format: 'Answer: \ $LETTER' where LETTER is one of ABCD."]
2 optimizer = TextualGradientDescent(parameters=[solution], constraints=
constraints)
```

Code Snippet 3 An example use for constraints when updating the solution to a problem.

Supplementary Note 2.3 Momentum

TextGrad supports the use of Momentum in the Textual Gradient Descent. In standard SGD momentum uses a linear combination of past gradients and the most recent one to define a new gradient to update a variable. Similarly, TextGrad keeps track of past iterations of the variable. This postfix is appended to the prompt for the optimizer.

```
1 optimizer = TextualGradientDescent(parameters=[solution], momentum_window=3)
```

Code Snippet 4 How to enable momentum using 3 previous steps in the TextualGradientDescent optimizer.

Momentum prompt

Here are the past iterations of this variable:
<PAST_ITERATIONS> {past_values} </PAST_ITERATIONS>

Supplementary Note 2.4 In-context examples

LLMs have shown an impressive ability to learn in-context from a few examples [1]. In-context examples can be utilized to improve textual gradients and update variables effectively. These examples serve as references to illustrate the characteristics of optimized variables. When in-context examples are applied, TextGrad adopts the following prompt to incorporate them:

In-context examples prompt

You must base on the following examples when modifying the {role_description}:
<EXAMPLES> {in_context_examples} </EXAMPLES>

By leveraging these examples, TextGrad can better understand and implement the properties of optimized variables, enhancing the overall optimization process.

Supplementary Note 3 Code optimization

Supplementary Note 3.1 Methodology

Baseline details. We re-run Reflexion [2] using the code by the authors currently available online. We had to make minor changes to ensure it ran correctly in our setting but we also contacted the original authors and ask feedback on our edits to ensure our evaluation was consistent. Moreover, we had to re-extract the LeetCodeHard [2] dataset using the authors’ pipeline; this means that it is likely that the dataset we are using in this manuscript is not the same dataset that was used in the Reflexion paper. While this dataset contains a set of simple tests to check if the code works as expected, the real evaluation in this context is given by passing the tests on the LeetCode platform.

Reflexion is run with a *one-shot prompt* that is meant to instruct the model on how to provide feedback. The pipeline run by Reflexion is as follows: given in input a prompt to generate code, a language model generates a first solution. If this solution passes the local tests, it is then submitted to the LeetCode platform to be evaluated on harder tests. However, if the solution does not pass the tests, we ask for feedback through Reflexion and optimize the code. Once again, if the new solution passes the local tests, we submit it to the LeetCode platform. We do this optimization for five iterations.

Example query for LeetCode Hard

```
def minimumTime(grid: List[List[int]]) -> int:
    """
```

You are given a 'm x n' matrix 'grid' consisting of non-negative integers where 'grid[row][col]' represents the minimum time required to be able to visit the cell '(row, col)', which means you can visit the cell '(row, col)' only when the time you visit it is greater than or equal to 'grid[row][col]'.

You are standing in the top-left cell of the matrix in the '0th' second, and you must move to any adjacent cell in the four directions: up, down, left, and right. Each move you make takes 1 second.

Return the minimum time required in which you can visit the bottom-right cell of the matrix. If you cannot visit the bottom-right cell, then return '-1'.

Example 1:

Input: grid = [[0,1,3,2],[5,1,2,5],[4,3,8,6]]

Output: 7

Explanation: One of the paths that we can take is the following:

- at t = 0, we are on the cell (0,0).
 - at t = 1, we move to the cell (0,1). It is possible because grid[0][1] <= 1.
 - at t = 2, we move to the cell (1,1). It is possible because grid[1][1] <= 2.
 - at t = 3, we move to the cell (1,2). It is possible because grid[1][2] <= 3.
 - at t = 4, we move to the cell (1,1). It is possible because grid[1][1] <= 4.
 - at t = 5, we move to the cell (1,2). It is possible because grid[1][2] <= 5.
 - at t = 6, we move to the cell (1,3). It is possible because grid[1][3] <= 6.
 - at t = 7, we move to the cell (2,3). It is possible because grid[2][3] <= 7.
- The final time is 7. It can be shown that it is the minimum time possible.

Example 2:

Input: grid = [[0,2,4],[3,2,1],[1,0,4]]

Output: -1

Explanation: There is no path from the top left to the bottom-right cell.

Constraints:

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

Supplementary Note 4 Solution optimization

```
1 # Assume we have the test_dataset
2 question, answer = test_dataset[i]
3 # Initialize the test time loss function
4 # This has the system prompt provided above as 'Solution Refinement
   Objective'
5 test_time_loss_fn = MultipleChoiceTestTime()
6 # Get a zero-shot solution from an LLM
7 solution: Variable = zero_shot_llm(question)
8 # Optimize the solution itself
9 optimizer = tg.TextualGradientDescent(parameters=[solution])
10
11 for iteration in range(max_iterations):
12     optimizer.zero_grad()
13     # Note how the loss is self-supervised (does not depend on any ground
       truth.)
14     loss = test_time_loss_fn(question, solution)
15     # Populate the gradients, and update the solution
16     loss.backward()
17     optimizer.step()
```

Code Snippet 5 A representative implementation of solution optimization in TextGrad.

Supplementary Note 4.1 Methods

For the CoT 0-shot prediction, we use the question template and system prompt released with GPT-4o in the [simple-evals](#) repository. In particular, to closely match their evaluations, we use the ChatGPT system prompt: “You are ChatGPT, a large language model trained by OpenAI, based on the GPT-4 architecture. \n Knowledge cutoff: 2023-12 \n Current date: 2024-04-01”. Further, we use the following template:

Multiple choice question answering template

Answer the following multiple choice question. The last line of your response should be of the following format: ‘Answer: \$LETTER’ (without quotes) where LETTER is one of ABCD. Think step by step before answering.

{Question}

A) {A}

B) {B}

C) {C}

D) {D}

During optimization, we provide the constraint to the optimizer that the prediction should conclude with an answer, following the simple-evals repository, through the following constraint description: “The last line of your response should be of the following format: ‘Answer: \$LETTER’ (without quotes) where LETTER is one of ABCD.”.

For solution refinement for multimodal question answering, we use the following objective:

Solution refinement objective for multimodal question answering

For visual mathematical reasoning:

LLM(“Please evaluate the existing answer to the visual math problem without solving it yourself. Verify that the answer provides accurate reasoning logic to address the question.

Image: {Image data}

Question: {Question}

Answer by the language model: {**Solution**}”)

For visual scientific reasoning:

LLM(“Please evaluate the existing answer to the visual scientific problem without solving it yourself. Verify that the answer accurately understands the image, provides appropriate knowledge and reasoning logic to address the question.

Image: {Image data}

Question: {Question}

Answer by the language model: {**Solution**}”

GPQA Diamond subset contains 198 questions. MMLU Machine Learning subset contains 112 questions, and College Physics subset contains 92 questions. At each iteration of optimization, we make one call to **gpt-4o** to evaluate the test time loss, one call to collect gradients, and one call to update the solution.

Supplementary Note 4.2 Prompts

Below is an example for GPQA Diamond task [3]:

Example query for GPQA Diamond

What is the concentration of calcium ions in a solution containing 0.02 M stoichiometric Ca-EDTA complex (we assume that the pH is ideal, $T = 25\text{ }^{\circ}\text{C}$). $\text{KCa-EDTA} = 5 \times 10^{10}$.

- A) $5.0 \times 10^{-3}\text{ M}$
- B) $1.0 \times 10^{-2}\text{ M}$
- C) $6.3 \times 10^{-7}\text{ M}$
- D) $2.0 \times 10^{-2}\text{ M}$

An example query for multimodal question answering benchmark, ScienceQA-IMG [4], is shown below:

Example query for ScienceQA-IMG

You will answer a scientific question based on an image. Please ensure you accurately interpret the image and think step by step. The last line of your answer should be formatted as follows: 'Answer: (X) Your Option.'

Context: Below is a food web from an ocean ecosystem in Monterey Bay, off the coast of California. A food web models how the matter eaten by organisms moves through an ecosystem. The arrows in a food web represent how matter moves between organisms in an ecosystem.

<image of a food web>

Question: In this food web, which organism contains matter that eventually moves to the sea cucumber?

Choices:

- (A) bat star
- (B) kelp
- (C) kelp bass
- (D) black rockfish

Reflexion baseline. In their closes question answering setting, [2] uses the ground truth supervision signal during test-time optimization. To make their approach applicable to solution optimization, we give the Reflexion algorithm access to our test-time objectives. In particular, in their reflection step, Reflexion uses our test time loss function and their own prompts to revise the solutions at test time. Otherwise, we are using their prompts and in-context examples shared in their [official repository](#). For multimodal reasoning and hallucination tasks, we prompt `gpt-4o` to generate image descriptions conditioned on the input query. This approach preserves the image’s informational content necessary for question answering while reducing the number of input tokens to fit within the input context window limit. We determine the optimal number of trials based on peak performance in the downstream task.

<image of an orange tree>

Question: An adult orange tree can grow seeds. Which part of the orange tree makes the seeds? (A) the flowers (B) the fruit

Solution at initialization:

The image shows an orange tree with numerous fruits and flowers, indicating different stages of the reproductive process. Flowers are pollinated, leading to the fertilization of ovules, which develop into seeds. These seeds are then encased within the fruit as it forms.

The flowers of the orange tree are pollinated, leading to the development of seeds within the ovules of the flowers. These seeds are then encased within the fruit as the flower turns into the fruit. **Therefore, the part of the orange tree that makes the seeds is the fruit. Answer: (B) the fruit**

Output of the loss function based on multimodal context:

The existing answer provides a detailed explanation of the biological process of seed formation in fruit trees, which is accurate. It correctly identifies that flowers are pollinated, leading to the development of seeds within the ovules, and these seeds are then encased within the fruit. The answer also correctly interprets the image, noting the presence of both fruits and flowers on the orange tree.

However, **there is a slight misinterpretation in the final conclusion.** While the seeds are indeed found within the fruit, **the part of the tree that initially makes the seeds is the flower, as the seeds develop from the fertilized ovules within the flowers.**

Solution after optimization:

The image shows an orange tree with numerous fruits and flowers, indicating different stages of the reproductive process. Flowers are pollinated, leading to the fertilization of ovules, which develop into seeds. These seeds are then encased within the fruit as it forms.

In summary, the flowers of the orange tree are pollinated, leading to the development of seeds within the ovules of the flowers. These seeds are then encased within the fruit as the flower turns into the fruit. **Therefore, the part of the orange tree that initially makes the seeds is the flower. Answer: (A) the flowers**

Supplementary Note 5 Prompt optimization

```
1 # Initialize the system prompt
2 system_prompt = tg.Variable(
3     "You are a helpful language model. Think step by step.",
4     requires_grad=True,
5     role_description="system prompt to the language model"
6 )
7
8 # Set up the model object 'parameterized by' the prompt.
9 model = tg.BlackboxLLM(system_prompt=system_prompt)
10
11 # Optimize the system prompt
12 optimizer = tg.TextualGradientDescent(parameters=[system_prompt])
13
14 for iteration in range(max_iterations):
15     batch_x, batch_y = next(train_loader)
16     optimizer.zero_grad()
17     # Do the forward pass
18     responses = model(batch_x)
19     losses = [loss_fn(response, y) for (response, y) in zip(responses,
20                                                            batch_y)]
21     total_loss = tg.sum(losses)
22     # Perform the backward pass and compute gradients
23     total_loss.backward()
24     # Update the system prompt
25     optimizer.step()
```

Code Snippet 6 A simple implementation of prompt optimization with TextGrad.

Supplementary Note 5.1 Tasks

Below, we provide an example query for each of the tasks in the prompt optimization section.

Example query for Word Sorting

Sort the following words alphabetically: oakland seaborg jacobi membrane trapezoidal allis marmot toggle anthology.

Example query for Object Counting

I have a couch, a bed, a car, a fridge, two tables, an oven, a toaster, and a chair. How many objects do I have?

Example query for GSM8k

Amber, Micah, and Ahito ran 52 miles in total. Amber ran 8 miles. Micah ran 3.5 times what Amber ran. How many miles did Ahito run?

For word sorting and object counting, we obtain the datasets from the [BBH repository](#), and we randomly split examples into 51 (training)/100 (validation)/99 (test)

samples. For GSM8k, we use the splits provided in DSPy [5] which has 200 (training)/300 (validation)/1,319 (test) samples.

We use the OPRO algorithm [6] shared in [their repository](#) with the default hyperparameters. In particular, the optimizer generates 8 candidates at every timestep, uses 3 in-context exemplars for the task with five optimizer iterations.

Evaluation. For object counting and GSM8k, we use the string-based exact match metric, which looks at the last numerical value provided in the answer, and compares it to the ground truth answer. For word sorting, we prompt `gpt-4o` to compare the ground truth list to the response provided in the answer, through the following prompt:

Evaluation system prompt for Word Sorting evaluation

System prompt:

Below is a question from a question-answering task, the ground truth answer, and reasoning with the final prediction. Is the final prediction correct, i.e. the same as the ground truth answer? Say only 1 (yes) or 0 (no). Return your response within `<ACCURACY>` `</ACCURACY>` tags. E.g., `<ACCURACY>` 0 `</ACCURACY>` or `<ACCURACY>` 1 `</ACCURACY>`.

Example prompt:

`**Question for the task:** {question}`
`**Ground truth answer:** {answer}`
`**Reasoning and prediction from the language model:** {prediction}`

Supplementary Note 6 Scalability and feasibility

Supplementary Note 6.1 Optimization using open-source models

We experiment with using open-source models in two different settings.

To provide further evidence on accessibility, we explore optimization using open-source models. In particular, we test prompt optimization using Llama-3.1-70B Instruct [11], which is a recently released open-source model. We use Llama-3.1-70B as an optimizer, and we test optimizing prompts for three different models: Llama-3.2-11B, Mixtral 8x22B [12], and Llama-3.1-70B itself. We make no changes to the framework for this optimization. We use the Llama turbo endpoints served by [Together AI](#). We use the `Meta-Llama-3.1-70B-Instruct-Turbo`, `Mixtral-8x22B-Instruct-v0.1`, `Llama-3.2-11B-Vision-Instruct-Turbo` endpoints for faster access to these models.

In Table [Supplementary Table 2](#), we show the results for prompt optimization. In particular, we see that prompts optimized for open-source models, using open-source models, can improve their performance. We observe that the degree of improvement is larger for more capable models. As open-source LLMs rapidly progress, we expect TextGrad to be even more useful to a broader audience.

Supplementary Table 1 Prompt optimization for reasoning tasks. With TextGrad, we optimize a system prompt or the in-context examples for `gpt-3.5-turbo` using `gpt-4o` as the gradient engine that provides the feedback during backpropagation.

Dataset	Method	Accuracy (%)
Object Counting [7]	CoT (0-shot) [8, 9]	77.8
	OPRO [6] (instruction-only, 0 demonstrations)	82.8
	DSPy (BFSR, 8 demonstrations) [5]	84.9
	TextGrad (3 demonstrations)	91.9
	TextGrad (instruction-only, 0 demonstrations)	91.9
	TextGrad (instruction and 3 demonstrations)	91.9
Word Sorting [7]	CoT (0-shot) [8, 9]	76.7
	OPRO [6] (instruction-only, 0 demonstrations)	77.8
	DSPy (BFSR, 8 demonstrations) [5]	79.8
	TextGrad (3 demonstrations)	78.8
	TextGrad (instruction-only, 0 demonstrations)	79.8
	TextGrad (instruction and 3 demonstrations)	80.8
GSM8k [10]	CoT (0-shot) [8, 9]	72.9
	OPRO [6] (instruction-only, 0 demonstrations)	76.8
	DSPy (BFSR, 8 demonstrations) [5]	81.1
	TextGrad (3 demonstrations)	80.1
	TextGrad (instruction-only, 0 demonstrations)	81.1
	TextGrad (instruction and 3 demonstrations)	82.4

Supplementary Note 6.2 Robustness of optimized prompts

To further the evidence on accessibility, we test the transferability of prompts optimized for `gpt-3.5-turbo`. In particular, we take the prompts optimized for `gpt-3.5-turbo`, and use them as prompts to various other open-source models. This is to demonstrate that once the optimization is done, without having to do it again, the optimized variables can transfer between different models.

We use Llama-3.1-8B Instruct and Llama-3.1-70B Instruct from [Together AI](#), Qwen-2.5 models [13] from Huggingface’s Transformers [14] library, and Gemini-1.5-Flash [15] from the Gemini family.

In Table [Supplementary Table 3](#), we present the results. Overall, prompts optimized for closed-source models using closed-source models transfer well to other models. In particular, we observe a greater degree of transferability for more capable models, and again we expect this degree of improvement to only get better as open-source models are becoming more capable.

Supplementary Note 6.3 Cost-accuracy analysis

We report the cost-accuracy frontier of prompt optimization in [Supplementary Fig. 1](#). In particular, we compare the fixed cost of optimizing a prompt for `gpt-3.5-turbo` using the existing techniques to the performance obtained via that method. This cost is paid once to optimize a prompt. On x-axis we report this cost, and on the y-axis, we report the performance.

Supplementary Table 2 Prompt optimization for reasoning tasks using open-source models. We optimize system prompts and in-context demonstrations for various models using TextGrad with Llama-3.1-70B Instruct as the gradient engine that provides feedback during backpropagation and the optimizer.

Dataset	Method Llama-3.2-11B Vision Instruct [11]	Accuracy (%)
Object Counting [7]	CoT (0-shot) [8, 9] TextGrad (instruction and 3 demonstrations)	82.8 89.4
Word Sorting [7]	CoT (0-shot) [8, 9] TextGrad (instruction and 3 demonstrations)	63.6 64.6
GSM8k [10]	CoT (0-shot) [8, 9] TextGrad (instruction and 3 demonstrations)	86.1 86.1
Mixtral 8x22B Instruct [12]		
Object Counting [7]	CoT (0-shot) [8, 9] TextGrad (instruction and 3 demonstrations)	87.9 89.8
Word Sorting [7]	CoT (0-shot) [8, 9] TextGrad (instruction and 3 demonstrations)	71.7 78.3
GSM8k [10]	CoT (0-shot) [8, 9] TextGrad (instruction and 3 demonstrations)	91.3 91.3
Llama-3.1-70B Instruct [11]		
Object Counting [7]	CoT (0-shot) [8, 9] TextGrad (instruction and 3 demonstrations)	85.9 93.1
Word Sorting [7]	CoT (0-shot) [8, 9] TextGrad (instruction and 3 demonstrations)	87.9 95.8
GSM8k [10]	CoT (0-shot) [8, 9] TextGrad (instruction and 3 demonstrations)	94.8 95.1

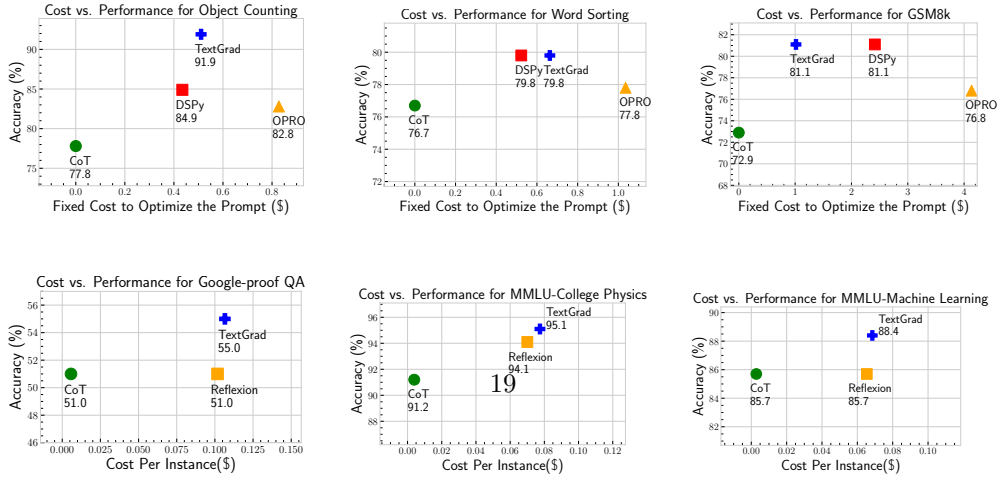
Overall, we highlight two observations. First, TextGrad consistently lies on the Pareto frontier. Second, the fixed cost of running prompt optimization is often less than \$1, highlighting that prompt optimization is not expensive to perform with these methods.

Secondly, we report the cost-accuracy frontier for solution optimization also in [Supplementary Fig. 1](#). Here, we report the average cost of running test-time optimization using `gpt-4o` for a given instance. This means for every test instance we run this algorithm, we incur this cost.

We again observe that TextGrad is at the Pareto frontier here, and the cost of running optimization per instance is usually less than \$0.10. This again highlights that running optimization using these models is often not expensive.

Supplementary Table 3 Transferability of optimized prompts. We test the transferability of prompts optimized for `gpt-3.5-turbo` using TextGrad on different models. Overall, optimized prompts improve the performance of various open-source and closed models, with up to 10% absolute performance gains.

Model	Dataset	Method	Accuracy
Gemini-1.5-Flash	GSM8k	CoT	91.1
		TextGrad-Transfer	92.3
	Object Counting	CoT	86.9
		TextGrad-Transfer	81.8
	Word Sorting	CoT	59.6
		TextGrad-Transfer	70.7
Qwen-2.5-14B Instruct	GSM8k	CoT	94.6
		TextGrad-Transfer	94.9
	Object Counting	CoT	85.9
		TextGrad-Transfer	86.9
	Word Sorting	CoT	32.3
		TextGrad-Transfer	33.3
Qwen-2.5-32B Instruct	GSM8k	CoT	95.6
		TextGrad-Transfer	96.6
	Object Counting	CoT	83.8
		TextGrad-Transfer	92.9
	Word Sorting	CoT	47.5
		TextGrad-Transfer	55.6
Llama-3.1-8B Instruct	GSM8k	CoT	86.8
		TextGrad-Transfer	86.8
	Object Counting	CoT	86.9
		TextGrad-Transfer	94.9
	Word Sorting	CoT	46.5
		TextGrad-Transfer	45.5
Llama-3.1-70B Instruct	GSM8k	CoT	94.4
		TextGrad-Transfer	94.0
	Object Counting	CoT	94.9
		TextGrad-Transfer	96.0
	Word Sorting	CoT	73.7
		TextGrad-Transfer	84.8



Supplementary Fig. 1 Cost-accuracy frontier for prompt optimization (top row) and instance optimization (bottom row). We observe that prompt optimization often has a cost less than \$1, and instance optimization often has a cost of less than \$0.1, highlighting that optimization with TextGrad is inexpensive.

Supplementary Note 7 Treatment plan optimization

Supplementary Note 7.1 Prompts

Radiotherapy treatment plan evaluation can be based on various dimensions, therefore there is no single score that can indicate the quality of plans. We adopt LLM to compute the “loss” by prompting it to assess the plan quality with clinical protocols. Specifically, LLM is used to compare each protocol with the current plan and produce the final assessment.

Supplementary Note 7.2 Inner-loop optimization for treatment planning

We employ a two-loop optimization approach [16], which includes i) an inner loop for inverse planning and ii) an outer loop for optimizing the hyperparameters of the inner loop. The inner loop focuses on traditional fluence map optimization, seeking to determine the optimal fluence map x by minimizing a cost function that combines multiple weighted objectives for various targets and organs at risk. This cost function is defined as:

$$\begin{aligned} \min_x \quad & \sum_{t=1}^{N_t} w_t ([Kx]_t - d_t)^2 + \sum_{s=1}^{N_s} w_s \Theta([Kx]_s - d_s) ([Kx]_s - d_s) \\ \text{s.t.} \quad & D_{95}([Kx]_t) = d_t, \\ & x \geq 0. \end{aligned} \tag{G7}$$

Here, $\{w_t\}_{t=1}^{N_t}$ and $\{w_s\}_{s=1}^{N_s}$ are the importance weights (the hyperparameters optimized by TextGrad) that balance the various objectives for N_t PTV targets and N_s OARs, respectively. K denotes the dose influence matrix, which specifies the dose per fluence unit delivered to each voxel in the volume by each beamlet. $\{d_t\}_{t=1}^{N_t}$ and $\{d_s\}_{s=1}^{N_s}$ are the scalar objective doses for each structure. The cost function essentially penalizes squared deviations from the target objective doses for the PTV targets and penalizes squared overdosing for the OARs only when doses exceed d_s . The Heaviside function Θ is used to ensure the objective considers only positive values. The minimization is constrained to positive x values and ensures that D_{95} — the minimum dose received by 95% of the structure volume — matches the prescribed dose for the clinical goal. We use `matRad` [17] with interior point algorithms to solve the inner-loop optimization.

In clinical practice, evaluating radiotherapy treatment plans involves balancing multiple quality metrics (e.g., mean dose, D_{95}), without a clear consensus on their relative importance. This complexity makes it difficult to design a unified “score function” that combines these metrics into a single objective for direct optimization. Additionally, many of these metrics are non-differentiable, which complicates the use of gradient-based optimization techniques. As a result, designing an effective optimization algorithm for the outer loop is both challenging and critical for successful automatic radiation oncology treatment planning.

Treatment plan loss prompt

Please act as an impartial and objective professional radiation oncologist. Your job is to evaluate the quality of radiation therapy treatment plans based on their dose volume histogram. You should consider the following protocols:
<Clinical Protocols>

Note that the order above does not indicate priority; always prioritize the regions that have protocols that are more significantly violated.

Here is the dose-volume histograms of the candidate plans for evaluation; each entry in the dose-volume histograms (DVH) table indicates the percentage of volume receiving a dose higher than a certain Gy (specified in the first column).

<DVH table>

We also provide the statistics of the above DVH table for ease of evaluation.
<DVH statistics>

Now, based on the protocols, and the DVH, please evaluate the plans. Avoid any positional biases and ensure that the order in which the responses are presented does not influence your decision. Be as objective as possible. Your answer must:

1. (Evidence) Extract the corresponding entries from the DVH table based on the factors to consider.
2. (Interpretation) Based on the extracted entries from DVH table, interpret them as: {percentage}% of volume encompassing the dose more than {dose} Gy for {organ}.
3. (Analysis) Combine the interpretation and the factors, analyze whether each of the numerical factors are met.

Based on the analysis on DVH tables, you need to produce a final evaluation. When all factors are satisfied, you should always answer no improvement is needed. If improvements are required, please provide suggestions on where to improve. Please ensure to follow the format below to return the evaluation results:

<FINAL> Decision: [The plan does/doesn't need to be improved.] Reasons: [list all factors that are not satisfied with detailed reasons, e.g. protocol X on (PTV/rectum/bladder/fh/body) is not satisfied because ...] </FINAL>

The final answer in the end must strictly follow the format above.

Previous works have typically used a weighted sum of multiple metrics, often referred to as a Plan Quality Metrics (PQMs), as the objective in the outer loop. Since some of these metrics are non-differentiable, [18] applied Bayesian Optimization (BO) to optimize the hyperparameters in the outer loop. Following their approach, we use Optuna [19] to implement Bayesian optimization for PQM as our baseline.

Supplementary Note 8 Molecule optimization

Supplementary Note 8.1 Molecule optimization

TextGrad supports a variety of optimization problems, including multi-objective optimization tasks commonly found in science and engineering applications. For example, in drug discovery, researchers seek to discover or design molecules that maximize a variety of objectives with regards to synthesizability, efficacy, and safety [20, 21]. To demonstrate TextGrad’s applications to multi-objective optimization, we apply TextGrad to drug molecule optimization, and show how our framework can interface with computational tools and optimize chemical structures towards simultaneously improving their binding affinity and druglikeness as measured by *in silico* oracles.

Task. A critical consideration for potential drug molecules is their *binding affinity*, which represents the strength of the interactions between the molecule and its protein target. Drug designers seek molecules with high binding affinities to relevant drug targets, as they require lower and less frequent doses to achieve efficacy. This affinity can be quantified by free energy ΔG , which describes the ratio of probabilities between bound and unbound ligand-receptor pairs. ΔG can be estimated using “docking” simulations of protein-ligand binding [22, 23]. In our experiments, we employ the Vina score from the Autodock Vina tool, a widely used physics-based docking simulator [24]. A more negative Vina score indicates that the drug is predicted to be more likely to bind to its intended target.

Potential drug molecules are also evaluated by their *druglikeness*, which estimates how the molecule will behave *in vivo*, with respect to solubility, permeability, metabolic stability and transporter effects. Molecules with high druglikeness are more likely to be absorbed by the body and reach their targets [25]. One metric for “druglikeness” is the Quantitative Estimate of Druglikeness (QED) score, a weighted composite score of chemical characteristics such as molecular weight, lipophilicity, polar surface area, among others. The QED score ranges from 0 to 1, where 1 indicates high druglikeness [26].

Though there are many more considerations for successful molecules, in our experiments, we restrict our objectives to minimizing the Vina score and maximizing the QED. Since each score evaluates different considerations in molecular design, optimizing one does not necessarily imply improvement in the other. Thus, optimizing molecules that simultaneously improve both objectives is non-trivial.

Methods. We apply TextGrad to drug molecule optimization by encoding molecules as SMILES strings and constructing a multi-objective loss from the Vina and QED scores. Namely, we perform test-time optimization over SMILES strings, where the gradients generated by TextGrad with respect to the multi-objective loss

are used to update the text representing the molecule:

$$\text{Evaluation} = \text{LLM}(\text{Affinity}(\text{SMILES}_i, \text{target}), \text{Druglikeness}(\text{SMILES}_i)), \quad (\text{H8})$$

$$\text{SMILES}_{i+1} = \text{TGD.step} \left(\text{SMILES}_i, \frac{\partial \text{Evaluation}}{\partial \text{SMILES}_i} \right). \quad (\text{H9})$$

We use `gpt-4o` as our LLM with the prompt text found in [Supplementary Note 8.3](#). At each iteration, the current molecule is evaluated by estimating its binding affinity to the target protein using the Vina score from Autodock Vina and using the QED score for druglikeness from RDKit ([Supplementary Note 8.2](#)). Each molecule is initialized as a small chemical fragment from a functional group. We apply TextGrad to all 58 targets in the DOCKSTRING molecule evaluation benchmark [27]. These 58 targets consist of clinically relevant proteins sampled from a variety of structural classes, 29 of which have clinically approved drugs. For each target, we optimize a starting fragment using TextGrad for up to 10 iterations ([Supplementary Note 8.8](#)), for 3 unique initial fragments. To evaluate our performance, we compare the characteristics of the molecules generated by TextGrad to clinically approved drugs for the respective protein ([Supplementary Note 8.4](#)).

Results. For all 58 targets, TextGrad consistently generates molecules with improved binding affinity and druglikeness irrespective of the initial starting fragment ([Supplementary Note 8.5](#)). For the 29 protein targets with clinically approved drugs, we observe that TextGrad generates molecules with highly competitive affinity and druglikeness when compared to clinical molecules evaluated using the same loss function ([Extended Data Fig. 1b](#)). The resulting molecules exhibit unique structures compared to their clinical approved counterparts and existing compounds ([Supplementary Note 8.6](#)), while maintaining similar *in silico* safety profiles ([Supplementary Note 8.7](#)). By combining specialized tools from computational chemistry with the general knowledge and reasoning capabilities of LLMs, TextGrad enables users to perform molecule optimization from arbitrary starting molecules, without the need for additional finetuning datasets, or high-performing initializations from curated compound libraries. Furthermore, TextGrad’s natural language gradients enable users to transparently analyze the optimization process and the impetus for the changes between each molecule in the trajectory. Though promising, we caution that the practical value of the molecules TextGrad has generated are limited. In particular, we optimize over and evaluate only a few properties, all evaluated using *in silico* objective functions (e.g. Vina, QED, ADMET scores), which are not reliable substitutes for experimental and clinical evaluation.

Supplementary Note 8.2 Docking and druglikeness evaluation

To optimize molecules, we evaluate the binding affinity and druglikeness of chemical structures encoded as SMILES strings. To compute both metrics, the generated SMILES string is first converted into an octet-complete Lewis dot structure using RDKit’s `MolFromSmiles` functionality. This method “sanitizes” molecules by adding

explicit hydrogens, kekulizing aromatic rings, standardizing valence states, and assigning radicals [28]. If this sanitization process fails at any step, whether through a structural ambiguity or an invalid molecule, the QED and Vina scores are replaced with a single string informing TextGrad that the molecule is invalid.

If the generated SMILES string does represent a valid chemical structure, we compute the QED score using RDKit’s `Chem.QED` function. While QED scores can be quickly and reliably computed, docking scores can vary significantly depending on target and ligand structure preparation. To ensure consistency, we calculate Vina scores using the DOCKSTRING package [27], which implements a standardized docking workflow and provides a set of gold-standard targets. In particular, after the ligand has been sanitized by RDKit, DOCKSTRING (de)-protonates it at pH 7.4 using Open Babel, and prepares and refines the 3D geometric structure using the Euclidean distance geometry algorithm ETKG and the classical force field MMFF94. Finally, Gasteiger charges are computed for all ligand atoms, and the resulting structure is saved as a ligand PDBQT file passed to Autodock Vina. For target preparation, the DOCKSTRING benchmark suite provides 58 curated crystal structures of clinically relevant proteins, with the majority at less than 2.5 Å resolution. These structures are specially prepared to improve correlations between theoretical and experimental binding affinities, for example by manual addition of polar hydrogens and removal of residual water and solute molecules. DOCKSTRING also standardizes simulation parameters such as numerical seeds, search box coordinates, and sampling exhaustivity, to ensure reproducible scoring.

Supplementary Note 8.3 Objective functions

Once the scores have been computed, they are passed to an LLM in the following format.

Molecule optimization prompt

Given a docking and a druglikeness score, and a molecule as a SMILES string provide a short criticism to improve the druglikeness of this molecule and its binding affinity with the protein {protein_name}. For docking, lower is better (less than -10 is considered good) and for druglikeness, 1 is the best and 0 is the worst (greater than 0.8 is considered good). In terms of prioritization, the docking score is {vina_qed_ratio} times as important as the druglikeness score. Make sure your criticism is very succinct and to the point.

```
# if smiles_string is valid
SMILES: {smiles_string}, Docking: {Vina}, Druglikeness: {QED}
```

```
# if smiles_string is invalid
SMILES: {smiles_string}, This molecule is invalid.
```

In practice, drug efficacy is typically a higher priority than absorption efficiency [29, 30], so we set the `vina_qed_ratio` to be 10. Empirically, we observe that by

scaling this prioritization factor, we can tune TextGrad’s generation towards molecules with differing binding affinity and druglikeness tradeoffs ([Supplementary Note 8.9](#)). This prioritization also allows us to simplify post-hoc selection and ranking of the generated molecules with a single “overall” score defined as follows, where a lower overall score s_{overall} indicates a better molecule:

$$s_{\text{overall}}(\text{molecule}, \text{protein}) = \text{Vina}(\text{molecule}, \text{protein}) + \frac{10}{\text{vina_qed_ratio}}(1 - \text{QED}(\text{molecule})). \quad (\text{H10})$$

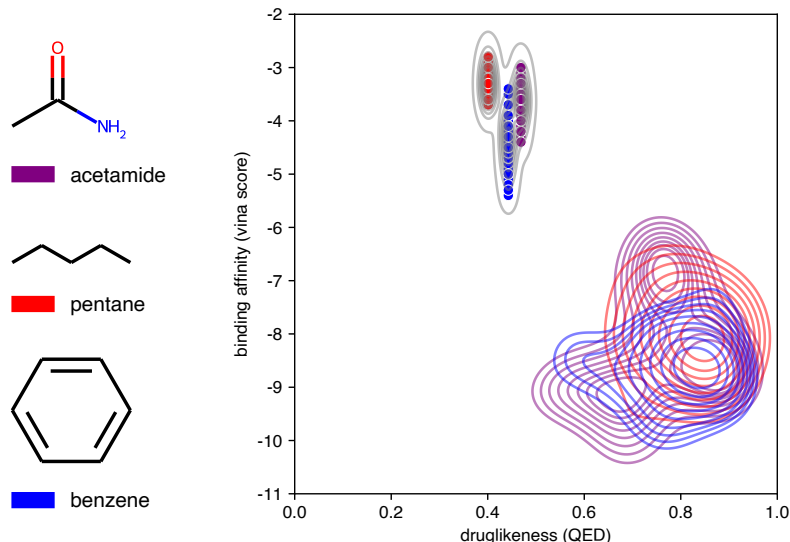
Since the QED score is bounded between 0 and 1, and the Vina score typically ranges between -3.0 to -12.0 kcal/mol, s_{overall} with the default `vina_qed_ratio` = 10 places approximately 10 times more emphasis on binding affinity than druglikeness as desired.

Supplementary Note 8.4 Benchmarks

To benchmark the performance of TextGrad generated molecules, we compare their characteristics to clinically approved drugs for the same protein targets found in DrugBank, a database of 16,619 drugs [31]. To ensure that the DrugBank molecules were both comparable and high quality, we filtered for drugs that were small molecules, had full clinical approval, and were designed for orthosteric binding with the same active site in the DOCKSTRING benchmark suite. After these filtering criteria, we identified 118 drugs targeting 29 of the 58 DOCKSTRING proteins. When evaluating binding affinity and druglikeness for the clinically approved drugs, we compute the Vina and QED scores using the same tools and workflow described in [Supplementary Note 8.2](#), exactly as applied to the TextGrad molecules.

Supplementary Note 8.5 Initialization

In practice, molecular optimization is typically accelerated by large scale pre-optimization screening, where libraries of millions or even billions of existing chemical structures are scored and ranked by docking, druglikeness, and other metrics. Only the most promising structures, termed “leads”, are further refined by medicinal chemists [32]. While TextGrad is capable of optimizing any initial molecule, in this work, we instead select our three initial molecules from simple fragments of common functional groups, shown in [Supplementary Fig. 2](#) in order to more accurately characterize its performance and avoid biasing its designs towards existing drugs. These fragments are highly diverse, and belong to different functional groups. Although these initial fragments have differing druglikeness and binding affinity characteristics, the impact of the starting fragment on TextGrad’s performance is minimal, and TextGrad is capable of generating molecules with high druglikeness and binding affinity from all three fragments.

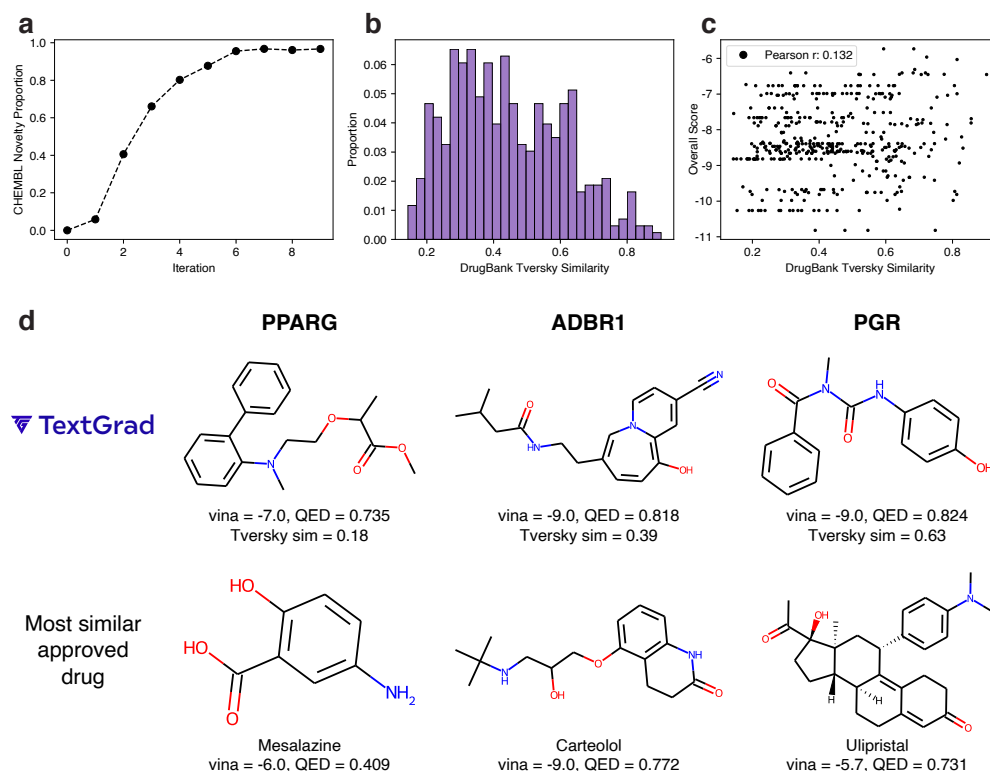


Supplementary Fig. 2 Molecule initialization. We initialize TextGrad with fragment molecules from three, diverse functional groups and optimize each initial molecule for 10 iterations for all 58 targets in the DOCKSTRING benchmark suite. For each fragment and each protein, we perform post selection on the generated molecules using the summary score outlined in equation H10 and visualize the resulting distribution. We observe that while the initial QED and Vina score distributions of the starting fragments vary greatly, the distribution of the optimized molecules is highly overlapping.

Supplementary Note 8.6 Chemical novelty

One of the primary concerns with using LLM models in scientific discovery is their capacity to simply memorize and regurgitate their training sets instead of performing logical reasoning [33]. Since recent models like `gpt-4o` are trained on massive, opaque datasets, a key concern is that the molecules TextGrad generates may simply be duplicates of clinically approved molecules for their respective targets. To quantify and test the extent or existence of this “memorization” effect, we compare the molecules that TextGrad generates with approved drugs for their respective protein targets, as well as unrelated but existing druglike chemical compounds.

We first compare TextGrad generated molecules for a particular protein target with all the clinically approved small molecules found in Drugbank for the same target (see Supplementary Note 8.4 for filtering criteria) using the Tversky similarity score on RDKit daylight chemical fingerprints. The Tversky similarity score is not symmetric, and compares substructures between a variant and reference molecule [34]. In our application, this is preferred as it assigns a high score to generated (variant) molecules that contain most or all of the substructures in the Drugbank (reference) molecule, even if there exist other extraneous substructures in the generated molecule that reduce the symmetric overlap between the two molecules. A Tversky score of 1.0 indicates that the TextGrad molecule is a complete subset of the Drugbank molecule, while 0.0 indicates no overlap.



Supplementary Fig. 3 Structural novelty. (a) We observe increasing novelty over optimization updates, where a generated molecule is considered “novel” if there does not exist any existing molecule in ChEMBL with a Tanimoto similarity score greater than 0.8. (b,c,d) We estimate substructure similarity using the Tversky metric between TextGrad molecules and clinically approved drugs for each target. While there are a wide range of similarity scores, we observe that similarity with known drugs has little correlation with molecule performance, suggesting that TextGrad’s generation process is weakly influenced by “memorized” knowledge of clinically approved structures.

In order to analyze the most relevant chemical structures, we restricted our analysis to a set of high performing TextGrad molecules, rather than all molecules from all iterations. In particular, using the overall score described in Equation H10, we selected the best molecule for each protein target and for each initial fragment, for a total of 87 generated molecules (29 druggable targets x 3 initial fragments) and 118 clinically approved drugs across 29 targets. For each target, we compute all pairwise Tversky scores between the 3 TextGrad molecules and the Drugbank molecules approved for that target, setting the Drugbank molecule as the reference, and the TextGrad molecule as the variant.

We observed that the distribution of Tversky similarity scores was quite broad, with a median of 0.42 but ranging from 0.14 to 0.90 (Supplementary Fig. 3b). However, we observe that Tversky similarity between TextGrad and DrugBank molecules is actually slightly anti-correlated with molecule performance as measured by the overall

score, suggesting that TextGrad’s optimization procedure is at best weakly influenced by prior knowledge of approved drugs. In fact, for a variety of generated molecules along a gradient of similarity scores, the generated molecules exhibit QED and Vina scores that match or even exceed their DrugBank counterparts (Supplementary Fig. 3d). We caution however that this does not imply that the TextGrad molecule holds clinical value, as clinically approved molecules are optimized on far more criteria, and our metrics of binding affinity and druglikeness are computed by *in silico* oracles that may not reflect experimental behavior.

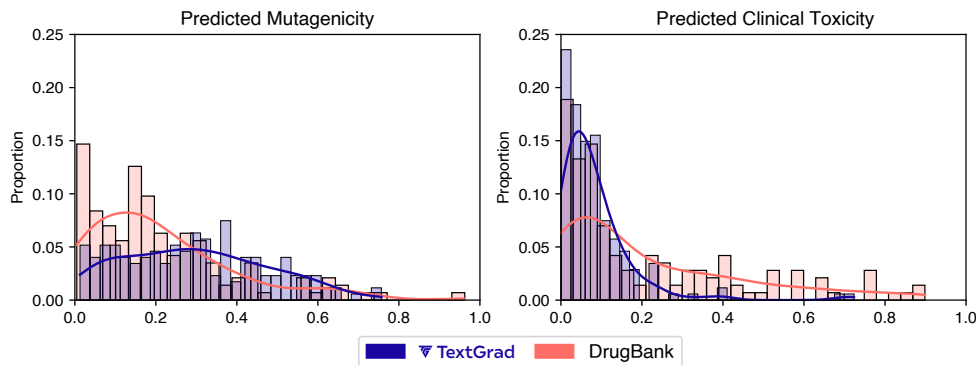
Beyond the similarity to known drugs, we are also interested in observing if TextGrad is generating novel molecules, or discovering previously unknown properties in existing compounds, for example a strong binding affinity to a protein target in a compound that has not been associated with the protein. To answer this question, we perform a Tanimoto similarity search across all of ChEMBL, a manually curated database of 2.4 million bioactive molecules with drug-like properties [35]. Unlike the Tversky score, the Tanimoto metric measures the symmetric overlap between two chemical structures, where 1.0 indicates an exact match, and 0.0 no overlap. We classify a TextGrad compound as *novel* if and only if there does not exist any molecule in ChEMBL with a Tanimoto similarity score over 0.80.

While the starting fragments are known molecules, we observe that as the number of iterations increases, TextGrad generates molecules that are progressively less likely to be previously known compounds. By the 6th iteration, 95% of all the molecules generated by TextGrad across all 58 targets and three starting fragments are novel using the criteria above (Supplementary Fig. 3a). By observing the trajectories of generated molecules and analyzing the textual gradients, we hypothesize that the feedback provided by the Vina and QED scores encourages TextGrad to explore chemical space beyond known molecules by progressively adding and removing combinations of functional groups. Since these chemical updates can form a combinatorial number of unique structures, it is reasonable that TextGrad would reach previously unexplored regions of chemical space in a relatively small number of iterations.

Supplementary Note 8.7 Extended post-generation analysis

We further evaluate our generated molecules post-generation, focusing on safety profiles related to mutagenesis and clinical toxicity. Mutagenicity refers to the ability of a drug to induce genetic alterations, which may lead to DNA damage and harmful long-term effects. Clinical toxicity refers to a broad range of adverse short or long-term side effects. Importantly, neither druglikeness nor binding affinity are strongly correlated with these criteria, and thus these desirability metrics are not directly optimized or otherwise explicitly encoded in TextGrad’s objectives.

To evaluate the propensity for mutagenesis and clinical toxicity, we employ the ADMET-AI model, that predicts these scores from the chemical structures of molecules [36]. ADMET-AI employs a deep learning model trained on multiple relevant datasets. In particular, for mutagenicity, ADMET-AI is trained on 7,255 drugs from the Ames dataset, a bacterial reverse mutation assay for rapidly screening large numbers of compounds for can induced genetic damage and frameshift mutations. A



Supplementary Fig. 4 Safety properties. We evaluate the predicted harmfulness of TextGrad molecules using the ADMET-AI model and compare them to clinically approved drugs. For both mutagenicity and clinical toxicity, 1.0 indicates a highly likelihood for harm and 0.0 a low likelihood. We observe that TextGrad proposes molecules a similar predicted distributions to clinical counterparts, despite neither metric being directly optimized.

predicted label of 1.0 indicates a high likelihood that the drug will induce mutagenesis, while a label of 0.0 indicates a low likelihood. For clinical toxicity, ADMET-AI is trained on the ClinTox dataset, a dataset of 1,484 drugs consisting of molecules that have failed clinical trials for toxicity reasons and also drugs that are associated with successful trials. Similarly, a predicted label of 1.0 indicates a high likelihood of clinical toxicity, while a label of 0.0 indicates a low likelihood.

Once again, we restrict our analysis to the best performing generated molecules with druggable targets as measured by the overall score, and select the best molecule for each protein target and for each initial fragment, for a total of 87 generated molecules. We then compare their predicted mutagenesis and clinical toxicity to the the 118 clinically approved molecules from DrugBank. We observe that for both Mutagenicity and Clinical Toxicity, the molecules generated by TextGrad have predicted distributions that indicate a low likelihood of harmful effects, and closely match the distributions of the clinically approved molecules. Together, these results show that the molecules generated by TextGrad have favorable predicted safety profiles, even though these criteria are not directly encoded in its loss function. While promising, we caution that these safety profiles do not guarantee that the TextGrad molecule is clinically safe, as we have only investigated two areas of toxicity both predicted by *in silico* oracles that are not guarantees of clinical behavior.

Supplementary Note 8.8 Parameter selection and early stopping

In the experiments outlined in [Supplementary Note 8.1](#), we employed early stopping to determine the number of optimization iterations. In particular, we score each molecule produced using a scoring function, and if the mean of the top- $K = 5$ scores does not improve by more than $\text{TOL} = 0.1$ for more than $\text{patience} = 5$ iterations, the run is

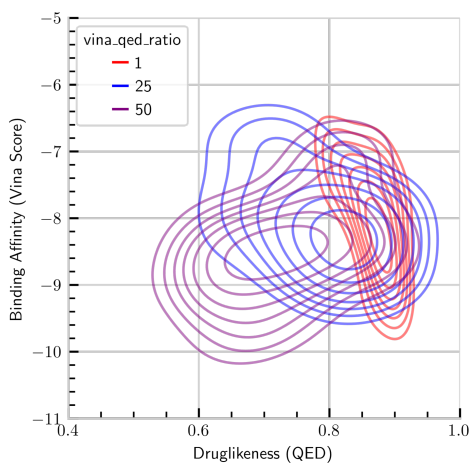
terminated, or until the `max_epoch = 10` is reached. Since our objective is both Vina and QED optimization, we use the overall score function defined in Equation H10. In general, a higher value of all the parameters (except `TOL`) will allow for longer explorations, and we invite users to select these parameters to suit their optimization budgets.

Supplementary Note 8.9 Multi-objective steerability

We perform additional experiments quantifying the affect of the `vina_qed_ratio` parameter in steering molecule property distributions in our multi-objective optimization experiment.

Using the prompt provided in [Supplementary Note 8.3](#), we set the `vina_qed_ratio` to 50, 25, and 1, respectively. This sequence of values represents an increase in the prioritization of the QED score relative to the Vina. Intuitively, for the value of 50, we essentially instruct TextGrad to disregard the QED score, while at 1, we inform TextGrad that the QED and Vina objectives are equally important. Starting from pentane, we optimize molecules for 10 iterations against all 58 proteins in the DOCKSTRING benchmark, for a total of 58 unique trajectories and 580 molecules.

We observe that tuning the `vina_qed_ratio` alters the distributions of the generated molecules and that an increase in QED prioritization led to a corresponding increase QED values. Conversely, a decrease in Vina prioritization led to a decrease in Vina performance (e.g. higher Vina scores). Using a paired t-test, we observed that the changes in both Vina and QED were statistically significant between the parameter settings of 50 and 1.



Supplementary Fig. 5 Vina/QED steerability. Distributions of molecule properties under different prioritization instructions. We observe a statistically significant worsening (increase) in the Vina scores upon reducing the Vina importance, and a corresponding significant improvement in the QED scores.

Supplementary Note 8.10 Gradient specificity analysis

We perform experiments to quantify the information content and specificity of TextGrad gradients. Concretely, we compare two optimization trajectories.

- **Original.** In the original TextGrad framework, an initial molecule is modified by computing the loss function against a particular protein and updating its structure based on the gradient computed with respect to the loss. This process is repeated for multiple iterations, with the all gradients computed against the same protein.
- **Random.** In the random framework, an initial molecule is modified by randomly selecting a gradient from a different molecule docked against a different protein. This process is repeated for the same number of iterations, but with random gradients selected at each iteration.

We evaluate both sequences of molecules using the loss function (QED and Vina Score) against the original protein. If the gradients produced by TextGrad are non-target specific, we expect that the randomized trajectory to perform as well as the original trajectory.

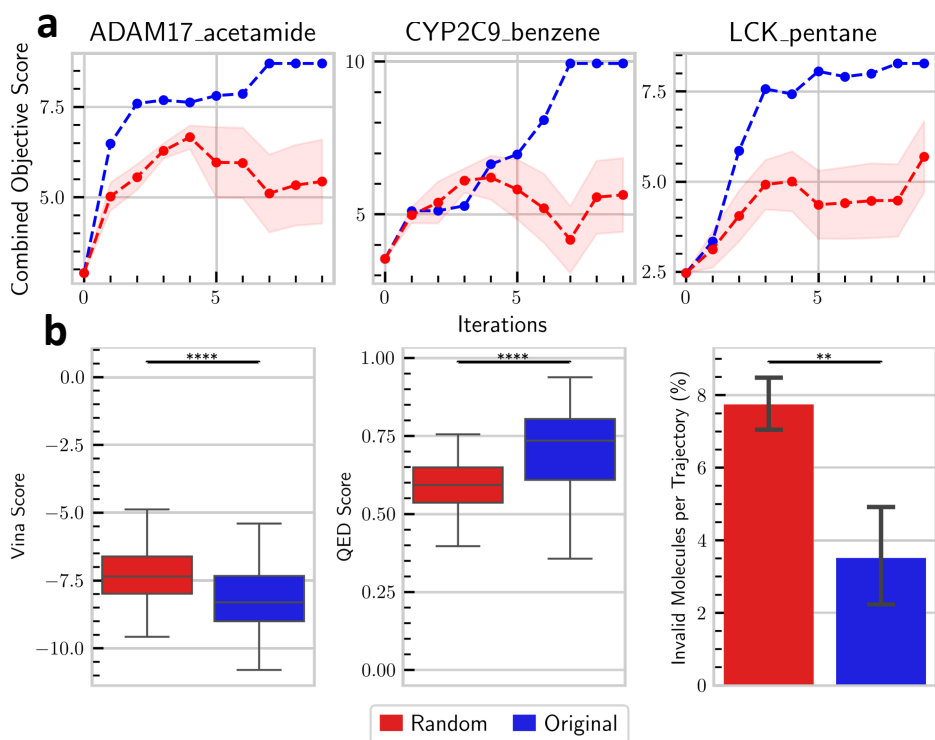
We compute "original" trajectories for all 58 targets in the DOCKSTRING benchmark, each starting from three initial starting fragments (benzene, pentane, acetamide) for a total of 174 original trajectories. For each original trajectory, we generate 10 random trajectories using the same initial fragment and evaluated against the same target, for a total of 1,740 random trajectories. Each trajectory, both random and original, was optimized for 10 iterations. While the optimization length was fixed to allow for a comprehensive comparison across multiple targets and for multiple randomizations, we observed that the trends for both the random and original trajectories are readily apparent within 10 iterations.

We observe that random trajectory produces lower performing molecules relative to original trajectory across both QED and Vina metrics. Using a paired t-test across protein targets and initial molecules, we compute the difference between the QED of the final molecule generated in the original trajectory and the mean QED value of the final molecules across the 10 repetitions in the random trajectories. For a total of $n = 174$ paired trajectory comparisons, we report that the random trajectories have statistically significant lower QED scores than the original trajectories with a p-value of less than $1e-4$. We repeat this comparison for the Vina score, and report similar results, with the random trajectories having significantly higher (worse) Vina scores ($p < 1e-4$). We also observe that random trajectories have a higher incidence of invalid molecules, producing on aggregate 1.5x more invalid SMILES strings than the original trajectory. Using the same statistical procedure for invalidity as the Vina and QED comparison, we report that the significance is $p < 1e-2$.

Supplementary Note 8.11 Additional benchmarks

We compare TextGrad to 6 molecular optimization algorithms, described briefly below.

- **Graph MCTS.** Based on Monte-Carlo Tree Search (MCTS), Graph MCTS applies MCTS to an atom-level molecular graph [37].



Supplementary Fig. 6 Gradient sensitivity analysis. (a) Sample performance trajectories under original and randomly selected gradients for different protein targets and initial molecules, using the combined objective score defined by Equation H10, with a `vina_qed_ratio` = 10, identical to the value in the loss prompt. (b) We observe that random trajectories underperform the original across Vina, QED, and percentage of valid molecules. Statistical significance computed using a paired t-test across protein-initial molecule pairs.

- **REINVENT.** Based on Reinforcement Learning (RL). REINVENT uses a policy based RL approach to tune RNNs to generate SMILES strings [38].
- **REINVENT-Selfies.** Similar to REINVENT, but with SELFIES strings instead of SMILES strings [39].
- **GP BO.** Combines Bayesian Optimization with a Gaussian Process kernel to sample molecular structures [40].
- **Smiles GA.** A genetic algorithm (GA) that explores molecular space through mutations and crossovers. In Smiles GA, the action space is taken over SMILES strings [41].
- **Graph GA.** Similar to Smiles GA, except that the action space is defined over molecular graphs [37].

Implementations were taken from the Practical Molecular Optimization (PMO) benchmark suite [39] without modification and with default hyperparameters, with

the exception of the Graph MCTS algorithm which accepts an initial molecule (default CC). For Graph MCTS, we replaced the default initial molecule with the same molecule used by TextGrad (pentane, acetamide, or benzene). For a full treatment of their implementation and algorithmic details, we refer the reader to the PMO benchmark and their original publications.

Objective. For this comparison, we modified our original multi-objective of optimizing both the Vina and QED scores to only the Vina score since the benchmark algorithms above only accept single-value objective functions. While it is possible to convert our original task of multi-objective Vina/QED optimization into a weighted single-value objective, we elected to simplify the comparison between algorithms and avoid having to arbitrarily select the QED importance factor.

Evaluation metrics. To compare the performances of different molecular optimization algorithms, we use the scaled AUC top- K metric introduced by the Practical Molecular Optimization benchmark [39], with $K = 5$. Briefly, the top- K trajectory measures the mean performance of the top K molecules generated at for a given oracle budget (in our setting, number of docking simulations). This is preferred over a single “best” ($K = 1$) molecule as it rewards algorithms that generate multiple viable candidates. The AUC of the top- K trajectory quantifies both the performance of the molecules generated as well the sample efficiency of the algorithm. In our setting, the objective function is the negative docking score (e.g. maximization) which is an unbounded objective that is variable between proteins. In order to compare algorithm performance across multiple proteins, we scale each trajectory to a $[0, 1]$ range by empirically deriving maximal and minimal AUCs for each protein across the set of algorithms compared. Specifically, for a given protein, we set the maximum AUC Top- K to be the AUC if the highest point in the top- K trajectory across all algorithms was discovered at the first oracle call, while the minimum to be the AUC if the lowest point in the top- k trajectory of all algorithms persisted for all oracle calls.

Set-up. We evaluated the 6 benchmark algorithms and TextGrad over all 58 unique protein targets in the DOCKSTRING benchmark dataset. For each protein target and algorithm, we repeated the optimization run three times for different randomization seeds for REINVENT, REINVENT-Selfies, Graph GA, Smiles GA, and GP BO. In the case of TextGrad and Graph MCTS, the runs were repeated for different initial molecules (benzene, pentane, acetamide). Each run was performed for a maximum oracle budget (e.g. number of docking simulations) of 50. For the benchmark algorithms, we ran all trajectories for the full 50 oracle calls. For TextGrad, in order to limit the number of LLM calls, we employed the early stopping procedure outlined previously. Briefly, if the mean top- $K = 5$ objective did not improve by more than $\text{TOL} = 0.1$ for more than $\text{patience} = 5$ iterations, the run was terminated. To calculate the AUC of these early stopped runs, we simply extend the last mean top- K objective score for the remaining oracle calls. Since the mean top- K is a non-decreasing function, this prevents us from overestimating TextGrad’s performance. In total, this experiment performed a comprehensive evaluation between the

7 algorithms across 58 proteins with 174 trajectories, and generating a total of 8,700 molecules.

Results. We outline the performance results across all 58 proteins in [Extended Data Fig. 2a](#) and sample trajectories in [Extended Data Fig. 2b](#). We conducted a paired t-test between the AUC Top- K of TextGrad for each protein against all other methods. The difference between TextGrad and all other methods was significant, with the greatest difference between TextGrad and Graph MCTCS and the closest between TextGrad and Graph GA. While these results demonstrate the promising potential of TextGrad to optimize molecules, we do not claim that TextGrad is the best method for this particular task, and have included these comparisons as a proof-of-concept. Furthermore, we reiterate that the practical value of TextGrad molecules is limited due to our reliance on *in silico* evaluation, and should not be interpreted as clinically useful.

Supplementary Note 9 Optimizing compound systems

Algorithm 2 Chameleon with single forward pass

```

1: Input: Query  $q$ , module inventory  $\mathcal{M}$ 
2: Output: Final response  $r$ 
3: # Initialize the cache
4:  $c^0 \leftarrow \emptyset$ 
5: # Generate the plan from the input query and modules
6:  $p \leftarrow \mathcal{P}(q; \mathcal{M})$ 
7: # Execute each module in the plan sequentially
8: for  $t = 1$  to  $T$  do
9:    $M^t \leftarrow \text{SelectModule}(p, t)$ 
10:   $y^t \leftarrow M^t(q; c^{t-1})$ 
11:   $c^t \leftarrow \text{update\_cache}(c^{t-1}, y^t)$ 
12: end for
13: # The final response is the output of the last module
14:  $r \leftarrow y^T$ 
15: Return  $r$ 

```

Details in optimizing the Chameleon agentic system. The `text_detector` tool in Chameleon was implemented using EasyOCR [42], an open-source OCR library. We set its `requires_grad` parameter to `False`, ensuring that this module’s output remains static during optimization. Consistent with the original Chameleon architecture, the planner module is a language model that operates without direct access to image inputs, maintaining a strict separation between planning and perception components. For comparison experiments, both the original and our TextGrad-optimized systems used the recent `gpt-4o-2024-08-06` version of GPT-4o and were evaluated on 400 test examples from ScienceQA-IMG.

Algorithm 3 TextGrad optimization for Chameleon

```
1: Input: Query  $q$ , module inventory  $\mathcal{M}$ , loss function  $\mathcal{L}$ , number of iterations  $N$ 
2: Output: Optimized final response  $r$ 
3: # Initialization
4:  $c^0 \leftarrow \emptyset$ 
5:  $p \leftarrow \mathcal{P}(q; \mathcal{M})$ 
6: # Initial forward pass
7: for  $t = 1$  to  $T$  do
8:    $M^t \leftarrow \text{SelectModule}(p, t)$ 
9:    $y^t \leftarrow M^t(q; c^{t-1})$ 
10:   $c^t \leftarrow \text{update\_cache}(c^{t-1}, y^t)$ 
11: end for
12:  $r \leftarrow y^T$ 
13: # Iterative chain optimization
14: for  $i = 1$  to  $N$  do
15:   for  $t = 1$  to  $T$  do
16:      $l \leftarrow \mathcal{L}(q, r)$  # Evaluate loss on the final response
17:      $g \leftarrow \text{gradient}(l, r)$  # Compute gradient w.r.t. the final response
18:      $g^t \leftarrow \text{backprop}(g, y^t)$  # Backpropagate gradient to module output
19:      $y^t \leftarrow \text{update}(y^t, g^t)$  # Refine the module output
20:     # Forward pass to update downstream modules
21:     for  $k = t + 1$  to  $T$  do
22:        $y^k \leftarrow M^k(q; c^{k-1})$ 
23:        $c^k \leftarrow \text{update\_cache}(c^{k-1}, y^k)$ 
24:     end for
25:      $r \leftarrow y^T$ 
26:   end for
27: end for
28: Return  $r$ 
```

Supplementary Note 10 Related Work

One related thread of work investigated the problem of prompt optimization. Practitioners demonstrated that prompt engineering strategies such as intelligently picking few-shot examples and in-context learning, CoT, ensembles can significantly boost performance of LLMs. To automate this process, white-box methods that leverage numerical gradients or probability distributions from LLMs were developed to optimize prompts [43–47], however, these methods cannot be used with closed-source models as they require access to model parameters. Other works investigated using LLMs as prompt optimizers [48–50].

Under prompt optimization, there are three works closest to our philosophy. First, DSPy [5] pioneered the idea of viewing complex LLM-based systems as programs with potentially many layers, and proposes ways to build and optimize them in a programmatic fashion. The framework is extensive, with results improving LLM performance in various question answering, reasoning, and prompt optimization tasks. Our work

takes a different perspective that backpropagation and its extensions can be a general and powerful framework to optimize the new generation of AI systems, and perform multiple tasks outside of prompt optimization. In particular, we treat not only instructions or demonstrations as variables to optimize, but also the instances we care about themselves — such as molecules, treatment plans, code snippets, and so on. Second, greatly inspiring to us, Prompt Optimization with Textual Gradients (ProTeGi) [50] defines the Textual Gradients in the context of prompt optimization, where gradients are natural language feedback from LLMs given to the mistakes made during the task. While ProTeGi is built on the textual gradient analogy, we expand this analogy more broadly to automatic differentiation, and going substantially beyond prompt optimization tasks. Third, OPRO [6] is one of the first papers to use LLMs as optimizers. OPRO uses an optimizer LLM to generate solution candidates, and uses the downstream task performance (called “score”) to iteratively select and improve available candidates. Importantly, since OPRO relies on a scalar supervision signal to select and improve solution candidates, it is not possible without further innovation to use it in settings such as instance optimization, or settings with multiple components where we use backpropagation. Furthermore, the implementation of the framework depends on the task, unlike the generality of our framework where we do not make any modifications to the implementation for new tasks. Finally, both DSPy, ProTeGi, and OPRO focused on prompt optimization as the key application, while a significant advance of TextGrad, as demonstrated through our diverse applications, is in instance optimization.

More generally, there is an emerging line of work built on the high-level idea of using LLMs as critics or optimizers [2, 6, 48–53]. While many of these earlier frameworks demonstrated the utility of LLMs as optimizers, we propose a single and general framework that was tested successfully in a variety of applications. Within this framework, we can reason about optimizing chains or stacks of LLMs: we propagate natural language feedback. Similarly, once viewed as a general-purpose optimization engine, we can formulate many relevant problems instantiated as a few lines of code in our framework, such as self-refinement of solutions [2, 54].

Building on the optimization analogy, we already transferred several analogies from the traditional optimization literature such as momentum [55] through using earlier iterations in the context, use of batch optimization [56], constrained optimization [57] using natural language constraints, and so on. Our work opens up a large space to design the new generation of optimization algorithms, all within the same framework.

Supplementary references

- [1] Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J.D., Dhariwal, P., Nee-lakantan, A., Shyam, P., Sastry, G., Askell, A., *et al.*: Language models are few-shot learners. In: Advances in Neural Information Processing Systems, vol. 33, pp. 1877–1901 (2020)
- [2] Shinn, N., Cassano, F., Gopinath, A., Narasimhan, K., Yao, S.: Reflexion: language agents with verbal reinforcement learning. In: Advances in Neural Information Processing Systems, vol. 36, pp. 8634–8652 (2023)
- [3] Rein, D., Hou, B.L., Stickland, A.C., Petty, J., Pang, R.Y., Dirani, J., Michael, J., Bowman, S.R.: GPQA: A graduate-level google-proof q&a benchmark. In: First Conference on Language Modeling (2023)
- [4] Lu, P., Mishra, S., Xia, T., Qiu, L., Chang, K.-W., Zhu, S.-C., Tafjord, O., Clark, P., Kalyan, A.: Learn to explain: Multimodal reasoning via thought chains for science question answering. In: Advances in Neural Information Processing Systems, vol. 35, pp. 2507–2521 (2022)
- [5] Khattab, O., Singhvi, A., Maheshwari, P., Zhang, Z., Santhanam, K., Haq, S., Sharma, A., Joshi, T.T., Moazam, H., Miller, H., *et al.*: DSPy: Compiling declarative language model calls into state-of-the-art pipelines. In: The Twelfth International Conference on Learning Representations (2024)
- [6] Yang, C., Wang, X., Lu, Y., Liu, H., Le, Q.V., Zhou, D., Chen, X.: Large language models as optimizers. In: The Twelfth International Conference on Learning Representations (2024)
- [7] Suzgun, M., Scales, N., Schärli, N., Gehrmann, S., Tay, Y., Chung, H.W., Chowdhery, A., Le, Q., Chi, E., Zhou, D., Wei, J.: Challenging BIG-bench tasks and whether chain-of-thought can solve them. In: Findings of the Association for Computational Linguistics: ACL 2023, pp. 13003–13051 (2023)
- [8] Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., Le, Q.V., Zhou, D., *et al.*: Chain-of-thought prompting elicits reasoning in large language models. In: Advances in Neural Information Processing Systems, vol. 35, pp. 24824–24837 (2022)
- [9] Kojima, T., Gu, S.S., Reid, M., Matsuo, Y., Iwasawa, Y.: Large language models are zero-shot reasoners. In: Advances in Neural Information Processing Systems, vol. 35, pp. 22199–22213 (2022)
- [10] Cobbe, K., Kosaraju, V., Bavarian, M., Chen, M., Jun, H., Kaiser, L., Plappert, M., Tworek, J., Hilton, J., Nakano, R., Hesse, C., Schulman, J.: Training verifiers to solve math word problems. arXiv preprint arXiv:2110.14168 (2021)

- [11] Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., Schelten, A., Yang, A., Fan, A., et al.: The llama 3 herd of models. arXiv preprint arXiv:2407.21783 (2024)
- [12] Jiang, A.Q., Sablayrolles, A., Roux, A., Mensch, A., Savary, B., Bamford, C., Chaplot, D.S., Casas, D.d.l., Hanna, E.B., Bressand, F., et al.: Mixtral of experts. arXiv preprint arXiv:2401.04088 (2024)
- [13] Yang, A., Yang, B., Hui, B., Zheng, B., Yu, B., Zhou, C., Li, C., Li, C., Liu, D., Huang, F., et al.: Qwen2 technical report. arXiv preprint arXiv:2407.10671 (2024)
- [14] Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., *et al.*: Transformers: State-of-the-art natural language processing. In: Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations, pp. 38–45 (2020)
- [15] Reid, M., Savinov, N., Teplyashin, D., Lepikhin, D., Lillicrap, T., Alayrac, J.-b., Soricut, R., Lazaridou, A., Firat, O., Schrittwieser, J., et al.: Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. arXiv preprint arXiv:2403.05530 (2024)
- [16] Xing, L., Li, J.G., Donaldson, S., Le, Q.T., Boyer, A.L.: Optimization of importance factors in inverse planning. *Physics in Medicine & Biology* **44**(10), 2525 (1999)
- [17] Wieser, H.-P., Cisternas, E., Wahl, N., Ulrich, S., Stadler, A., Mescher, H., Müller, L.-R., Klinge, T., Gabrys, H., Burigo, L., *et al.*: Development of the open-source dose calculation and optimization toolkit matrad. *Medical Physics* **44**(6), 2556–2568 (2017)
- [18] Wang, Q., Wang, R., Liu, J., Jiang, F., Yue, H., Du, Y., Wu, H.: High-dimensional automated radiation therapy treatment planning via bayesian optimization. *Medical Physics* **50**(6), 3773–3787 (2023)
- [19] Akiba, T., Sano, S., Yanase, T., Ohta, T., Koyama, M.: Optuna: A next-generation hyperparameter optimization framework. In: Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 2623–2631 (2019)
- [20] Nicolaou, C.A., Brown, N.: Multi-objective optimization methods in drug design. *Drug Discovery Today: Technologies* **10**(3), 427–435 (2013)
- [21] Hoelder, S., Clarke, P.A., Workman, P.: Discovery of small molecule cancer drugs: successes, challenges and opportunities. *Molecular Oncology* **6**(2), 155–176 (2012)

- [22] Kontoyianni, M.: Docking and virtual screening in drug discovery. *Proteomics for Drug Discovery: Methods and Protocols*, 255–266 (2017)
- [23] Agarwal, S., Mehrotra, R.: An overview of molecular docking. *JSM Chemistry* **4**(2), 1024–1028 (2016)
- [24] Trott, O., Olson, A.J.: AutoDock Vina: improving the speed and accuracy of docking with a new scoring function, efficient optimization, and multithreading. *Journal of Computational Chemistry* **31**(2), 455–461 (2010)
- [25] Ursu, O., Rayan, A., Goldblum, A., Oprea, T.I.: Understanding drug-likeness. *Wiley Interdisciplinary Reviews: Computational Molecular Science* **1**(5), 760–781 (2011)
- [26] Bickerton, G.R., Paolini, G.V., Besnard, J., Muresan, S., Hopkins, A.L.: Quantifying the chemical beauty of drugs. *Nature Chemistry* **4**(2), 90–98 (2012)
- [27] García-Ortegón, M., Simm, G.N., Tripp, A.J., Hernández-Lobato, J.M., Bender, A., Bacallado, S.: DOCKSTRING: easy molecular docking yields better benchmarks for ligand design. *Journal of Chemical Information and Modeling* **62**(15), 3486–3502 (2022)
- [28] RDKit: Open-source cheminformatics. <http://www.rdkit.org>
- [29] Hughes, J.P., Rees, S., Kalindjian, S.B., Philpott, K.L.: Principles of early drug discovery. *British Journal of Pharmacology* **162**(6), 1239–1249 (2011)
- [30] Wenlock, M.C., Austin, R.P., Barton, P., Davis, A.M., Leeson, P.D.: A comparison of physiochemical property profiles of development and marketed oral drugs. *Journal of Medicinal Chemistry* **46**(7), 1250–1256 (2003)
- [31] Knox, C., Wilson, M., Klinger, C.M., Franklin, M., Oler, E., Wilson, A., Pon, A., Cox, J., Chin, N.E., Strawbridge, S.A., *et al.*: Drugbank 6.0: the drugbank knowledgebase for 2024. *Nucleic Acids Research* **52**(D1), 1265–1275 (2024)
- [32] Berry, M., Fielding, B., Gamielien, J.: Practical considerations in virtual screening and molecular docking. *Emerging Trends in Computational Biology, Bioinformatics, and Systems Biology*, 487 (2015)
- [33] Birhane, A., Kasirzadeh, A., Leslie, D., Wachter, S.: Science in the age of large language models. *Nature Reviews Physics* **5**(5), 277–280 (2023)
- [34] Nikolova, N., Jaworska, J.: Approaches to measure chemical similarity—a review. *QSAR & Combinatorial Science* **22**(9-10), 1006–1026 (2003)
- [35] Gaulton, A., Bellis, L.J., Bento, A.P., Chambers, J., Davies, M., Hersey, A., Light, Y., McGlinchey, S., Michalovich, D., Al-Lazikani, B., *et al.*: ChEMBL: a large-scale bioactivity database for drug discovery. *Nucleic Acids Research* **40**(D1),

1100–1107 (2012)

- [36] Swanson, K., Walther, P., Leitz, J., Mukherjee, S., Wu, J.C., Shivnaraine, R.V., Zou, J.: ADMET-AI: A machine learning ADMET platform for evaluation of large-scale chemical libraries. *Bioinformatics* **40**(7), 416 (2024)
- [37] Jensen, J.H.: A graph-based genetic algorithm and generative model/monte carlo tree search for the exploration of chemical space. *Chemical Science* **10**, 3567–3572 (2019)
- [38] Olivecrona, M., Blaschke, T., Engkvist, O., Chen, H.: Molecular de-novo design through deep reinforcement learning. *Journal of Cheminformatics* **9**, 1–14 (2017)
- [39] Gao, W., Fu, T., Sun, J., Coley, C.: Sample efficiency matters: a benchmark for practical molecular optimization. In: *Advances in Neural Information Processing Systems*, vol. 35, pp. 21342–21357 (2022)
- [40] Tripp, A., Simm, G.N., Hernández-Lobato, J.M.: A fresh look at de novo molecular design benchmarks. In: *NeurIPS 2021 AI for Science Workshop* (2021)
- [41] Yoshikawa, N., Terayama, K., Sumita, M., Homma, T., Oono, K., Tsuda, K.: Population-based de novo molecule generation, using grammatical evolution. *Chemistry Letters* **47**(11), 1431–1434 (2018)
- [42] AI, J.: EasyOCR (2024). <https://jaided.ai/easyocr/>
- [43] Shin, T., Razeghi, Y., Logan IV, R.L., Wallace, E., Singh, S.: AutoPrompt: Eliciting knowledge from language models with automatically generated prompts. In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pp. 4222–4235 (2020)
- [44] Jia, M., Tang, L., Chen, B.-C., Cardie, C., Belongie, S., Hariharan, B., Lim, S.-N.: Visual prompt tuning. In: *European Conference on Computer Vision*, pp. 709–727 (2022)
- [45] Li, X.L., Liang, P.: Prefix-tuning: Optimizing continuous prompts for generation. In: *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pp. 4582–4597 (2021)
- [46] Chen, X., Zhang, N., Xie, X., Deng, S., Yao, Y., Tan, C., Huang, F., Si, L., Chen, H.: Knowprompt: Knowledge-aware prompt-tuning with synergistic optimization for relation extraction. In: *Proceedings of the ACM Web Conference 2022*, pp. 2778–2788 (2022)
- [47] Sordoni, A., Yuan, E., Côté, M.-A., Pereira, M., Trischler, A., Xiao, Z., Hosseini, A., Niedtner, F., Le Roux, N.: Joint prompt optimization of stacked llms using

- variational inference. In: Advances in Neural Information Processing Systems, vol. 36 (2024)
- [48] Zhou, Y., Muresanu, A.I., Han, Z., Paster, K., Pitis, S., Chan, H., Ba, J.: Large language models are human-level prompt engineers. In: The Eleventh International Conference on Learning Representations (2023)
 - [49] Ye, Q., Ahmed, M., Pryzant, R., Khani, F.: Prompt engineering a prompt engineer. In: Findings of the Association for Computational Linguistics: ACL 2024, pp. 355–385 (2024)
 - [50] Pryzant, R., Iter, D., Li, J., Lee, Y., Zhu, C., Zeng, M.: Automatic prompt optimization with “gradient descent” and beam search. In: Bouamor, H., Pino, J., Bali, K. (eds.) Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, pp. 7957–7968 (2023)
 - [51] Song, X., Tian, Y., Lange, R.T., Lee, C., Tang, Y., Chen, Y.: Position: Leverage foundational models for black-box optimization. In: Forty-first International Conference on Machine Learning (2024)
 - [52] Xiao, T.Z., Bamler, R., Schölkopf, B., Liu, W.: Verbalized machine learning: Revisiting machine learning with language models. arXiv preprint arXiv:2406.04344 (2024)
 - [53] Gao, Z., Du, Y., Zhang, X., Ma, X., Han, W., Zhu, S.-C., Li, Q.: Clova: A closed-loop visual assistant with tool usage and update. In: Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, pp. 13258–13268 (2024)
 - [54] Madaan, A., Tandon, N., Gupta, P., Hallinan, S., Gao, L., Wiegrefe, S., Alon, U., Dziri, N., Prabhume, S., Yang, Y., *et al.*: Self-refine: Iterative refinement with self-feedback. In: Advances in Neural Information Processing Systems, vol. 36 (2024)
 - [55] Polyak, B.T.: Some methods of speeding up the convergence of iteration methods. USSR Computational Mathematics and Mathematical Physics **4**(5), 1–17 (1964)
 - [56] Robbins, H., Monro, S.: A stochastic approximation method. The Annals of Mathematical Statistics, 400–407 (1951)
 - [57] Boyd, S., Boyd, S.P., Vandenberghe, L.: Convex optimization. Cambridge University Press (2004)