北京大学
PEKING UNIVERSITY

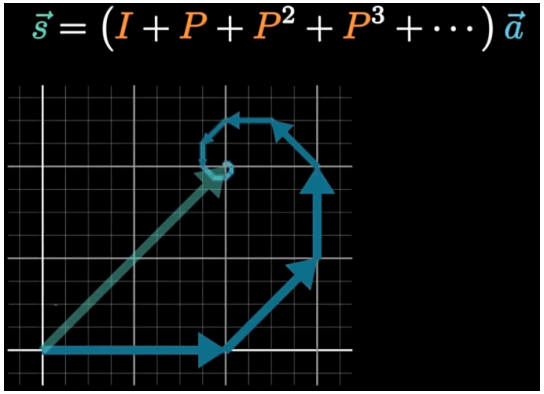
TextGrad

arXiv https://arxiv.org/pdf/2406.07496v1

Jiayu Yao

Operator Algebra & Group Theory

$$s = \sum_{k=0}^{\infty} a x^k = \frac{a}{1-x}$$ 等比数列求和



$$s = (I - P)^{-1} a \leftarrow a = s\,(I - P)$$ 广义特征算子

$$e^x = \int e^x$$

$$\rightarrow \left(I - \int\right) e^x = 0$$

$$\rightarrow e^x = \left(I - \int\right)^{-1} 0$$

$$\rightarrow e^x = \left(I + \int + \int^2 + \cdots\right) 0 = C + Cx + \frac{C}{2}x^2 + \frac{C}{6}x^3 + \cdots$$

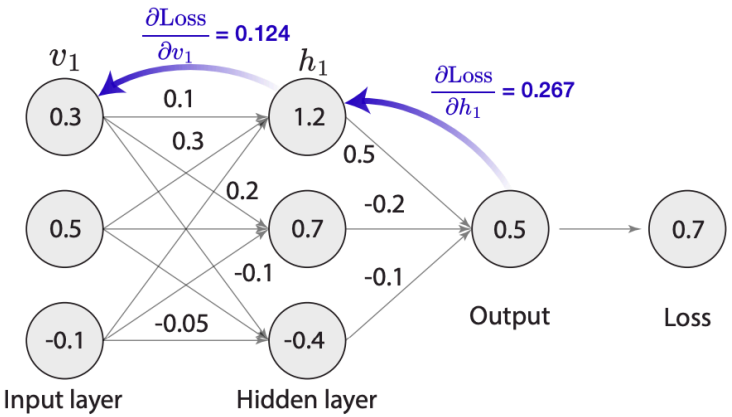
泰勒展开

Mathematics is the art of giving the same name to different things

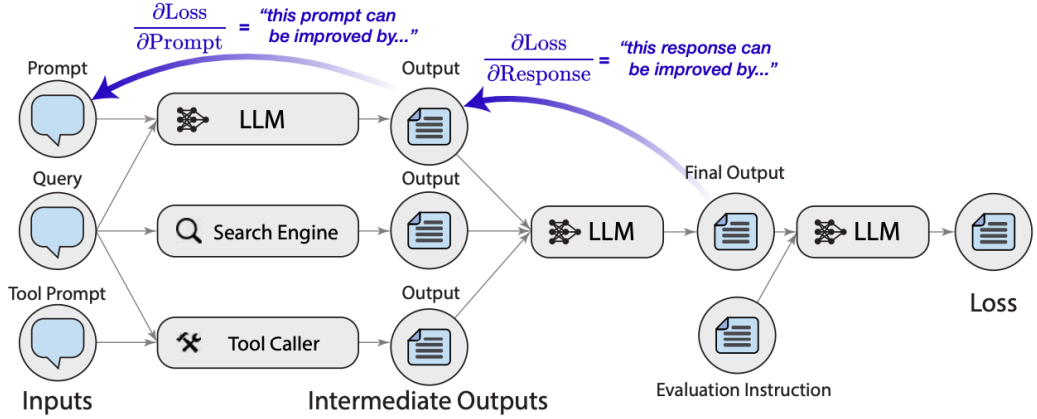
—— Henri Poincaré

TextGrad

a **Neural network and backpropagation using numerical gradients**



b **Blackbox AI systems and backpropagation using natural language 'gradients'**



c **1 Analogy in abstractions**

	Math	PyTorch	TextGrad
Input	x	`Tensor(image)`	`tg.Variable(article)`
Model	$\hat{y} = f_\theta(x)$	`ResNet50()`	`tg.BlackboxLLM("You are a summarizer.")`
Loss	$L(y, \hat{y}) = \sum_i y_i \log(\hat{y}_i)$	`CrossEntropyLoss()`	`tg.TextLoss("Rate the summary.")`
Optimizer	$\text{GD}(\theta, \frac{\partial L}{\partial \theta}) = \theta - \frac{\partial L}{\partial \theta}$	`SGD(list(model.parameters()))`	`tg.TGD(list(model.parameters()))`

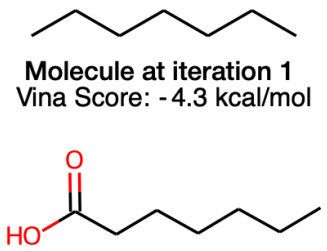
2 Automatic differentiation

PyTorch and TextGrad share the same syntax for backpropagation and optimization.

Forward pass	Backward pass	Updating variable
`loss = loss_fn(model(input))`	`loss.backward()`	`optimizer.step()`

TextGrad

d TextGrad for molecule optimization



Gradients
Add functional groups that increase polarity for stronger interactions.

Gradients
Introduce polar or aromatic groups to increase interaction points.

e TextGrad for code optimization

```
for i in range(n):
    if nums[i] < k:
        balance -= 1
    elif nums[i] > k:
        balance += 1
    if nums[i] == k:
        result += count.get(balance, 0) +
            count.get(balance - 1, 0)
    else:
        result += count.get(balance, 0)
    count[balance] = count.get(balance, 0) + 1
```

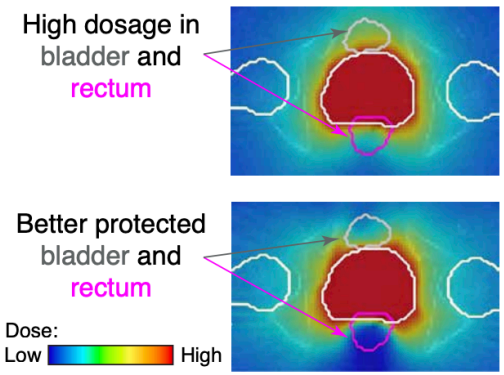
Code at iteration t

```
for i in range(n):
    if nums[i] < k:
        balance -= 1
    elif nums[i] > k:
        balance += 1
    else:
        found_k = True
    if nums[i] == k:
        result += count.get(balance, 0) +
            count.get(balance - 1, 0)
    else:
        count[balance] = count.get(balance, 0) + 1
```

Code at iteration t+1

Gradients
Handling `nums[i] == k`: The current logic does not correctly handle the case when `nums[i] == k`. The balance should be reset or adjusted differently when `k` is encountered. ...

f TextGrad for treatment plan optimization



Gradients
The current weight for the rectum and bladder are relatively low, which is not sufficient to protect the rectum and bladder...

g TextGrad for prompt optimization

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.

Prompt at initialization (Accuracy = 77.8%)

You will answer a reasoning question. List each item and its quantity in a clear and consistent format, such as '- Item: Quantity'. Sum the values directly from the list and provide a concise summation. Ensure the final answer is clearly indicated in the format: 'Answer: $VALUE' where VALUE is a numerical value. Verify the relevance of each item to the context of the query and handle potential errors or ambiguities in the input. Double-check the final count to ensure accuracy."

Prompt after optimization (Accuracy = 91.9%)

Law of Gradient Chain

Gradient computation for the simple graph: Prompt $\xrightarrow{\text{LLM}}$ Prediction $\xrightarrow{\text{LLM}}$ Evaluation

$$\frac{\partial \text{Evaluation}}{\partial \text{Prediction}} = \nabla_{\text{LLM}}(\text{Prediction}, \text{Evaluation}), \tag{4}$$

$$\frac{\partial \text{Evaluation}}{\partial \textbf{Prompt}} = \frac{\partial \text{Evaluation}}{\partial \text{Prediction}} \circ \frac{\partial \text{Prediction}}{\partial \textbf{Prompt}} = \nabla_{\text{LLM}}(\textbf{Prompt}, \text{Prediction}, \frac{\partial \text{Evaluation}}{\partial \text{Prediction}}), \tag{5}$$

where we use ∇_{LLM} for the gradient operator when the forward function is an LLM call.[a] In particular, this function returns natural language feedback such as '*This prediction can be improved by...*' where the feedback describes how to modify the variable to improve the downstream objective, analogous to gradients in optimization. We first collect feedback to the Prediction variable using the evaluation. Then, given this feedback and the (**Prompt** $\xrightarrow{\text{LLM}}$ Prediction) call, we collect the feedback on the **Prompt**.

[a] Note that we overload the notation to denote both for cases when the output variable does not have successors (e.g., $\nabla_{\text{LLM}}(\text{Prediction}, \text{Evaluation})$) and when it has successors, and thus gradients (e.g., $\nabla_{\text{LLM}}(\textbf{Prompt}, \text{Prediction}, \frac{\partial \text{Loss}}{\partial \text{Prediction}})$).

Law of Gradient Chain

Example implementation for the gradient operator

$$\frac{\partial \mathcal{L}}{\partial x} = \nabla_{\text{LLM}}\left(x, y, \frac{\partial \mathcal{L}}{\partial y}\right) \triangleq \texttt{"Here is a conversation with an LLM: \{x|y\}."} \tag{6}$$

$$+$$

$$\text{LLM}\big(\texttt{Here is a conversation with an LLM: \{x|y\}.}$$

$$\texttt{Below are the criticisms on \{y\}:}$$

$$\left\{\frac{\partial \mathcal{L}}{\partial y}\right\}$$

$$\texttt{Explain how to improve \{x\}.}\big),$$

where the gradient object is a combination of the context in which the variable appears and the feedback obtained from an LLM[a], defined analogously to Pryzant *et al.* [25]. Note that this operator does not depend on, e.g., the application domain. Once implemented, the gradient operator for LLM calls are fixed throughout the framework for all applications.

[a] The exact prompts we use are different to ensure generality and flexibility; we use these examples only for exposition.

Law of Gradient Chain

Example optimizer. In standard gradient descent, the current value of the variable is combined with the gradients through subtraction, e.g.:

$$\theta_{\text{new}} = \text{GradientDescent.step}(\theta, \frac{\partial \mathcal{L}}{\partial \theta}) = \theta - \frac{\partial \mathcal{L}}{\partial \theta}. \tag{7}$$

Continuing the gradient-based optimization analogy, we use *Textual* Gradient Descent (TGD):

Updating the Prompt variable in the simple graph via TGD.

$$\textbf{Prompt}_{\textbf{new}} = \text{TGD.step}(\textbf{Prompt}, \frac{\partial \text{Evaluation}}{\partial \textbf{Prompt}}). \tag{8}$$

to update the parameters, in this case, the **Prompt**. In particular, given the current variable and the gradients (feedback) we collected for this variable, the optimizer seeks to update this variable.

Law of Gradient Chain

A concrete way to instantiate TGD is the following:

Example implementation of one TGD iteration

$$x_{\text{new}} = \text{TGD.step}\left(x, \frac{\partial \mathcal{L}}{\partial x}\right) \triangleq \text{LLM}\Big(\texttt{Below are the criticisms on \{x\}:} \quad \left\{\frac{\partial \mathcal{L}}{\partial x}\right\} \quad \texttt{Incorporate the criticisms, and produce a new variable.}\Big). \tag{9}$$

where x is the variable we would like to improve, and $\frac{\partial \mathcal{L}}{\partial x}$ is the feedback we obtained for the variable during the backward pass[a]. Similar to the gradient operator, this function also does not depend on the domain of application, and TGD implementation is the same across all uses of the framework.

[a] The exact prompts we use are different to ensure generality and flexibility; we use these examples only for exposition.

Batch TextGrad

$$v = f_v(\text{PredecessorsOf}(v)) \quad \forall v \in \mathcal{V}$$

$$\frac{\partial \mathcal{L}}{\partial v} = \bigcup_{w \in \text{SuccessorsOf}(v)} \nabla_{\text{f}} \left(v, w, \frac{\partial \mathcal{L}}{\partial w} \right),$$

Experiments

Table 2: Solution optimization for zero-shot question answering with `gpt-4o`.

Dataset	Method	Accuracy (%)
Google-proof QA [27]	CoT [46, 47]	51.0
	Best reported [48]	53.6
	TEXTGRAD	**55.0**
MMLU-Machine Learning [45]	CoT [46, 47]	85.7
	TEXTGRAD	**88.4**
MMLU-College Physics [45]	CoT [46, 47]	91.2
	TEXTGRAD	**95.1**

Experiments

Table 3: Prompt optimization for reasoning tasks. With TEXTGRAD, we optimize a system prompt for `gpt-3.5-turbo` using `gpt-4o` as the gradient engine that provides the feedback during backpropagation.

Dataset	Method	Accuracy (%)
Object Counting [50, 51]	CoT (0-shot) [46, 47]	77.8
	DSPy (BFSR, 8 demonstrations) [10]	84.9
	TEXTGRAD (instruction-only, 0 demonstrations)	**91.9**
Word Sorting [50, 51]	CoT (0-shot) [46, 47]	76.7
	DSPy (BFSR, 8 demonstrations) [10]	**79.8**
	TEXTGRAD (instruction-only, 0 demonstrations)	**79.8**
GSM8k [52]	CoT (0-shot) [46, 47]	72.9
	DSPy (BFSR, 8 demonstrations) [10]	**81.1**
	TEXTGRAD (instruction-only, 0 demonstrations)	**81.1**

Example: TextGrad optimized prompt for `gpt-3.5-turbo-0125`

Prompt at initialization (GSM8k Accuracy= 72.9%):
You will answer a mathematical 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."

Prompt after 12 iterations with batch size 3 (GSM8k Accuracy= 81.1%):
You will answer a mathematical reasoning question. Restate the problem in your own words to ensure understanding. Break down the problem into smaller steps, explaining each calculation in detail. Verify each step and re-check your calculations for accuracy. Use proper mathematical notation and maintain consistency with the context of the question. Always conclude with the final answer in the following format: 'Answer: $VALUE' where VALUE is a numerical value.

Thanks