# AVIARY: TRAINING LANGUAGE AGENTS ON CHALLENGING SCIENTIFIC TASKS

March 31, 2025

## 1 Stochastic Computation Graphs

In the general case, a language agent may include both stochastic and deterministic operations.

We build on the formalism of stochastic computation graphs (SCG) [1]: directed, acyclic graphs with nodes corresponding to computations and edges corresponding to arguments.

A deterministic node $v$ corresponds to a function $f_v$, and the node's output $o(v)$ is defined as:

$$o(v) = f_v(\{o(w) \,|\, w \in \text{parents}(v)\}) \tag{1}$$

Similarly, a stochastic node $u$ is defined by a (conditional) distribution $p_u$, with output:

$$o(u) \sim p_u(\cdot \,|\, \{o(w) \,|\, w \in \text{parents}(v)\}) \tag{2}$$

Note that inputs to the graph can be treated as constant (deterministic) nodes. Outputs of the graph are leaf nodes.

A language agent's policy is simply an SCG with a string input (the observation) and a string output (the action). Language agent architectures can be easily expressed as SCGs by combining deterministic and stochastic nodes. The SCGs of the below common language agent architectures are visualized in Figure 1.

(a) Language model as policy: a single stochastic node corresponding to sampling from the language model.

(b) Retrieval-augmented generation (RAG): a deterministic node (document retrieval) leading to a stochastic node (LLM sampling).

(c) Rejection sampling from LLM: several stochastic nodes (LLM samples), all leading to a deterministic node (selecting the preferred sample).

(d) ReAct [2]: two consecutive stochastic nodes, corresponding to sampling a reasoning string and an action (tool call).

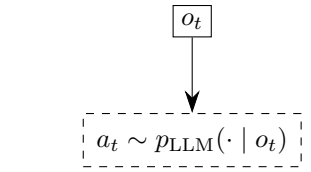
In many cases, a language agent defines an agent state $\xi_t$ that is a function of previous observations and actions. For example, the agent state of a multi-turn LLM conversation is typically defined as:

$$\begin{aligned} \xi_t &= [o_0, a_0, \ldots, a_{t-1}, o_{t-1}] \\ a_t &\sim p_{\text{LM}}(\cdot|\xi_t) \end{aligned} \tag{3}$$

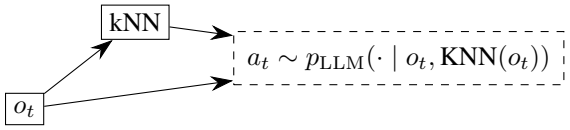
The SCG output is then a tuple of $(a_t, \xi_{t+1})$, namely an action and a new state. Separating the state enables batching of agent states and observations, as well as keeping the SCG as a true function. Memories, if desired, are considered part of the agent state, and their retrieval is incorporated in the SCG.

### 1.1 Gradient Estimation

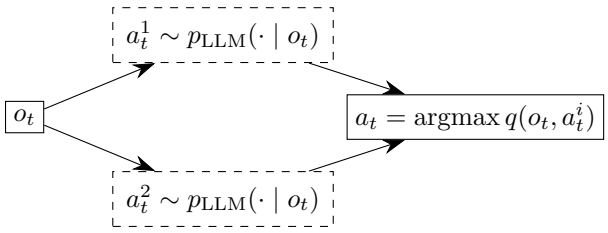
Aviary defines an SCGs to allow for the automatic computation of unbiased gradient estimates through the graph. We follow the approach outlined in the original SCG paper for computing these estimates [1]. Our framework introduces its own backpropagation method, which handles stochastic nodes and even nodes that require calling external tools. When



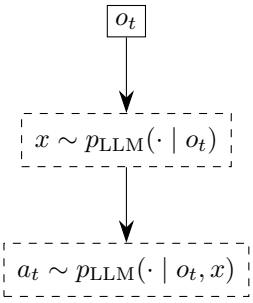

(a) Language model as policy




(b) Retrieval-augmented generation




(c) Rejection sampling from language model




(d) ReAct

Figure 1: Simple language agent architectures represented as stochastic computation graphs. Deterministic nodes are solid rectangles; stochastic nodes are dashed. Note that we augment the graphs with a deterministic input node to indicate how the observation $o_t$ is consumed.

passing through nodes that involve PyTorch operations, it falls back to `torch.autograd` [3], allowing our SCGs to leverage PyTorch's existing capabilities for deterministic nodes.

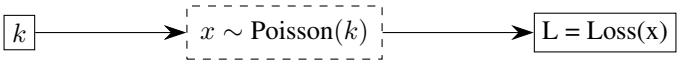
Our framework allows for the inclusion of approximate gradients when an analytic value cannot be computed. This flexibility is particularly useful in scenarios where the exact gradient cannot be calculated, such as when interacting with black-box APIs, which are prevalent in language agents.

### 1.2 SCG Experiments

We validate the parameter learning capability of our framework's SCG through two experiments.

### 1.3 Poisson Gradient Estimation

In the first experiment, we define a graph where a value $k$ is fed into a Poisson distribution, from which we sample a random variable $x$. The sampled variable $x$ is then passed to a loss module that computes a deterministic loss $L(x)$. Our goal is to learn the optimal value of $k$ that minimizes this loss. That will happen at some target value $t$. In this example, the loss function is $L = |x - \text{t}|$. The computational graph and the gradient formulation are as follows:




$$\nabla_k \mathbb{E}_{x \sim \text{Poisson}(k)}[L] = \mathbb{E}_{x \sim \text{Poisson}(k)}\left[L \cdot \nabla_k \ln P(x \mid k)\right] \qquad \text{(REINFORCE estimator)}$$

The analytical derivation of the gradient of the graph is the following:

$$
\begin{aligned}
&\nabla_k \mathbb{E}_{x\sim\text{Poisson}(k)}[L] = \nabla_k \sum P(x \mid k) \cdot L && \text{(by definition of expectation)}\\
&= \sum (\nabla_k P(x \mid k) \cdot L + P(x \mid k) \cdot \nabla_k L) && \text{(product rule for differentiation)}\\
&= \sum (P(x \mid k) \cdot \nabla_k L + L \cdot \nabla_k P(x \mid k)) && \text{(reordering terms for readability)}\\
&= \sum P(x \mid k) \cdot (\nabla_k L + L \cdot \nabla_k \ln P(x \mid k)) && \text{(using } \nabla_k P(x \mid k) = P(x \mid k) \cdot \nabla_k \ln P(x \mid k))\\
&= \mathbb{E}_{x\sim\text{Poisson}(k)} \left[\cancel{\nabla_k L} + L \cdot \nabla_k \ln P(x \mid k)\right] && \text{(since } L \text{ does not directly depend on } k)\\
&= \mathbb{E}_{x\sim\text{Poisson}(k)} \left[L \cdot \nabla_k \ln P(x \mid k)\right]
\end{aligned}
$$

Aviary allows to use the analytical gradients to do backpropagation through the graph. The loss function used is our experiment is $L(x) = |x - t|$ where $t$ is the target value. We use an SGD optimizer, a learning rate of 0.01, and a batch size of 8.

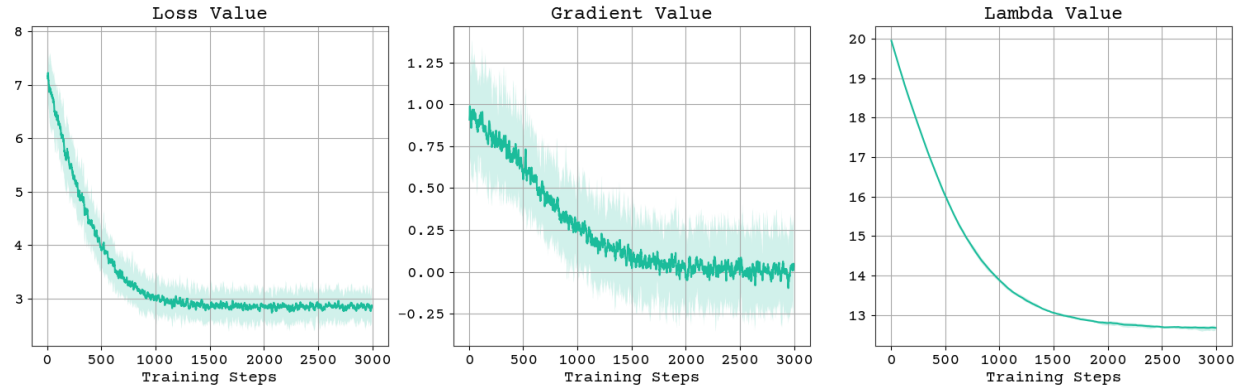
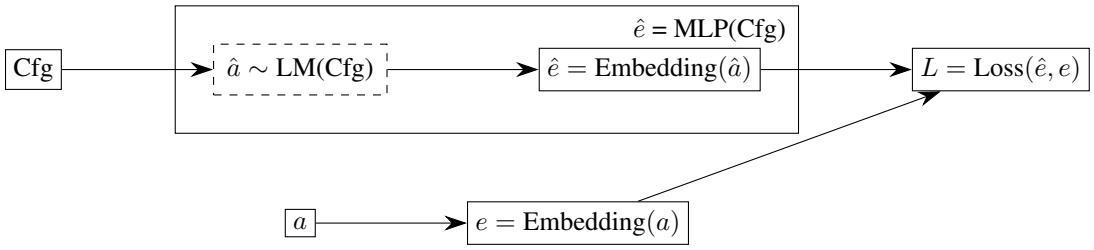



Figure 2: Poisson example optimization curves. The target value $t$ is 13 and the initial $k$ value is 20. We ran the experiment 5 times and report the mean, with shaded areas representing the maximum and minimum values observed.

### 1.4 Black Box Gradient Estimation

Our library also allows optimization of parameters through stochastic black-box nodes. We demonstrate this by optimizing the token limit of a closed language model API in a simple language task. In this example, we ask the model to write a specific word 10 times, with the expected. We deliberately set the maximum number of allowed output tokens in the API call to 4, which is too low to generate the full response. For the model to produce the correct output with 10 repeats, at least 20 tokens would be necessary. This token limit parameter is defined in an API configuration (Cfg). The computation graph we want to optimize is defined as follows:

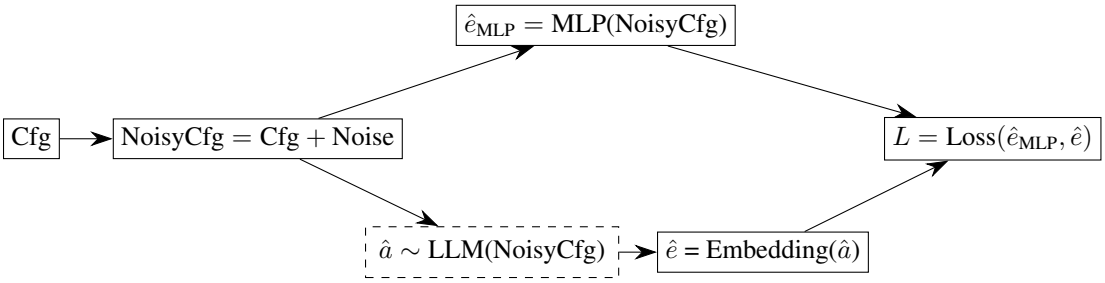



Since the language model is a stochastic node, an analytic gradient for the token limit cannot be computed using traditional automatic differentiation or standard stochastic computation graphs. However, our framework supports the use of gradient estimates when available. To obtain these estimates, we model the behavior of the language model and embedding nodes around the current configuration values with a Multi-Layer Perceptron (MLP) and backpropagate through it. This allows us to approximate the gradient for the token limit parameter contained in the Cfg as follows:

$$\nabla_{\text{Cfg}}\mathbb{E}[L] = \mathbb{E}\left[L\nabla_{\text{Cfg}} \ln P(\hat{e} \mid \text{Cfg})\right] \approx \mathbb{E}\left[\frac{\partial L}{\partial \hat{e}}\frac{\partial \hat{e}}{\partial \text{Cfg}}\right] = \mathbb{E}\left[\frac{\partial L}{\partial \hat{e}}\frac{\partial \text{MLP(Cfg)}}{\partial \text{Cfg}}\right]$$

The analytical derivation of the API call's configuration (Cfg) gradient for the Black Box is analogous to that of the Poisson graph, but we cannot compute it because API calls to language models typically return only the output value. However, if we obtain a good enough approximation, our framework allows it to be used in the backward pass.

To approximate the gradients, we train a Multi-Layer Perceptron (MLP) to model the behavior of the LLM given the Cfg using supervised learning. Noise (±2 token limit) is added to the configuration, and we predict the embedded LLM output. Mean Squared Error (MSE) loss is used to train the MLP. The diagram below illustrates this process.




For simplicity, we omit the following detail in the graph: the Cfg actually contains both the prompt and the parameter to be optimized, which is the token limit. We embed the prompt and concatenate it with the token limit value before passing it to the MLP. When computing the gradient, we only compute it with respect to the token limit parameter.

To account for changes in LLM behavior after updating configuration values, we alternate between 10 training updates for the MLP and 10 updates to the configuration values. Initially, we perform 300 MLP training steps. We introduce noise of ±2 tokens to explore the effects of different token limits while we do not change the prompt. The language model used is OpenAI's `gpt-4o-mini-2024-07-18`. The text embeddings have a size of 512, and they are obtained using OpenAI's `text-embedding-3-small` model. The MLP has 2 hidden layers, each with a size of 512, followed by layer normalization [4] and a final layer that predicts the mean and standard deviation of the output. We use the reparameterization trick to sample the output. An Adam optimizer [5] is used to train the MLP with a learning rate of 0.0005, while a simple SGD optimizer is used to train the token limit parameter with a learning rate of 10.

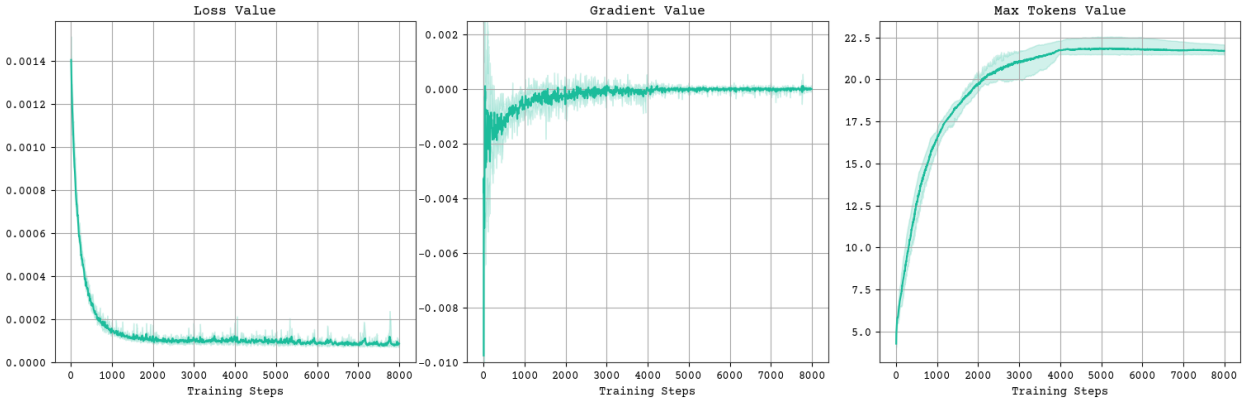



Figure 3: Black box optimization curves. The loss decreases rapidly and converges near zero. The gradient decreases to zero as the learning process stabilizes. The max token value increases from the initial value of 4 to over 20, then stabilizes as gradients diminish. Shaded areas indicate minimum and maximum values across 5 random seeds.

## References

[1] John Schulman, Nicolas Heess, Theophane Weber, and Pieter Abbeel. Gradient estimation using stochastic computation graphs. *Advances in Neural Information Processing Systems*, 28, 2015. 1

[2] Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: synergizing reasoning and acting in language models. In *International Conference on Learning Representations (ICLR)*, 2023. 1

[3] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in PyTorch. In *NIPS 2017 Workshop on Autodiff*, 2017. 2

[4] Jimmy Lei Ba. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016. 4

[5] Diederik P Kingma. Adam: A method for stochastic optimization. In *Third International Conference on Learning Representations*, 2015. 4