Reinforcement Learning for Online Adaptive Optimization

Phuong-Mai Huynh-Pham [*1,2] **Stefan M. Wild** [2]

Abstract

Hyper-parameter selection is a crucial challenge for most of the optimization algorithms. Additionally, using static hyper-parameters instead of adaptively adjusting them in an online manner may hinder the overall progress of the algorithm. Therefore, in this work, we explore the use of reinforcement learning (RL) techniques for the adaptive selection of the hyper-parameters in derivative-free optimization (DFO) settings. Firstly, as a test case, we focus on dynamic policies for the RASTA randomized DFO algorithm. Secondly, we use RL techniques to optimize the parameters of a policy function which is represented as a neural network. The resulting policy function is used online during optimization process to choose the hyper-parameters adaptively. The preliminary results show the efficacy of our method: our framework is able to choose the action hyper-parameters that in turns increases the reward values.

1. Introduction & Related works

Optimization algorithms, especially those for randomized, large-scale optimization often have a number of internal algorithmic parameters that govern the mechanics of the algorithm. Often default values for these are obtained by running the algorithm on a test-set of problems and doing very coarse optimization or doing optimization for optimization (e.g., BFO (Porcelli & Toint, 2017)).

There has been several works in developing a framework for large-scale stochastic derivative-free optimization (DFO) (Audet & Hare, 2018) (Larson et al., 2019), mostly focusing on the trust-region method, such as STARS (Dzahini & Wild, 2022), ASTRO-DF (Shashaani et al., 2018), RSDFO (Cartis & Roberts, 2021). In this work, we explore the use of RL techniques for the adaptive selection of the hyper-parameters in DFO settings. We introduce RASTA algorithm (short for reinforcement-learning adaptive stochastic trust-region algorithm), a further development of STARS (Dzahini & Wild, 2022). First of all, as a test case, we focus on dynamic policies for the RASTA randomized DFO algorithm. Secondly, we use RL techniques to optimize the parameters of a policy function which is represented as a neural network (NN). The resulting policy function is used online during the optimization process to choose the hyper-parameters adaptively.

[*]Equal contribution [1]Center for Applied Mathematics, Cornell University, Ithaca, New York, USA [2]Lawrence Berkeley National Laboratory, Berkeley, California, USA. Correspondence to: Phuong-Mai Huynh-Pham <ph463@cornell.edu>.

Final Project for CS 6787 - Advanced Machine Learning System, Cornell University, Ithaca, New York, Spring 2024.

2. Methodology

2.1. Notations

- $\mathcal{S}$ denotes the set of all possible states (continuous), and $\mathcal{A}$ is the set of all possible actions (discrete). Consequently, $\boldsymbol{s}_t \in \mathcal{S}$ is the state and $\boldsymbol{a}_t \in \mathcal{A}$ is the action at time step t.
- $\pi(\boldsymbol{s}_t)$ is the policy function. If we have a stochastic policy, then $\boldsymbol{a}_t \sim \pi(\boldsymbol{s}_t)$, or if we have a deterministic policy, then $\boldsymbol{a}_t = \pi(\boldsymbol{s}_t)$. In our setting, the policy function is deterministic, and it is represented using an NN which is parameterized by $\boldsymbol{w}$. Hence, the policy function is $\pi_{\boldsymbol{w}}(\boldsymbol{s}_t)$.
- The reward at time t is $r_t = R(\boldsymbol{s}_t, \boldsymbol{a}_t)$ where $R : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is the reward function.
- $J(\boldsymbol{s}_t)$ is the value function.
- Discount factor for the value function: $\gamma \in (0, 1)$. The discount factor plays a role (weights of reward) in deciding the policy, i.e how much is going to be rewarded given the action taking at that state.

Our state vector will take the form

$$\begin{aligned}\boldsymbol{s}_t = [&t,\ h_t,\ \dots h_{t-T+1}, \eta_t,\ \dots \eta_{t-T+1}, b_t,\ \dots b_{t-T+1},\\ &\Delta_t,\ \dots \Delta_{t-T+1}, \rho_t,\ \dots \rho_{t-T+1},\\ &f(\boldsymbol{x}_{k-1}),\ \dots f(\boldsymbol{x}_{k-T}), \boldsymbol{x}_k,\ \dots \boldsymbol{x}_{t-T+1}]\end{aligned} \tag{1}$$

where b_t is a binary (0,1) variable indicating whether iteration t was successful.

Our state transition is via $\boldsymbol{s}_t = \text{Iterator}(\boldsymbol{s}_{t-1})$. Agent is NN and environment is the iterator

2.2. Objective function

Our program aims to minimize the following quadratic optimization problem, presented in stochastic form:

$$\min_{\boldsymbol{x}\in\mathbb{R}^{d_x}} f(\boldsymbol{x}) = \mathbb{E}_{\boldsymbol{\zeta}}\left[\frac{1}{2}\|\boldsymbol{L}^T(\boldsymbol{x}-\bar{\boldsymbol{x}})+\boldsymbol{\zeta}\|^2\right] \tag{2}$$

where $\boldsymbol{\zeta}_i \sim \mathcal{N}(0,\alpha^2)$ are i.i.d. for $i = 1, ..., d_x$, for a set of $\alpha \geq 0$ and where $\nabla_{\boldsymbol{x}} f$ is not available. Here, $\boldsymbol{L}$ can be thought as a generic loss function.

2.3. RASTA algorithm

Algorithm 1 Simplified RASTA Iteration

Input: Starting point $\boldsymbol{x}_0 \in \mathbb{R}^{d_x}$, initial trust region radius $\Delta_0 > 0$, and subspace dimension $p \in \{1, 2, \cdots, d_x\}$
Hyper-parameters: Acceptance thresholds $0 < \eta_k < 1$ and finite difference parameter $h_k > 0$, for $k = 0, 1, 2, ...$

for $k = 0, 1, 2, \ldots$ **do**
  Define a subspace by randomly sampling

$$\boldsymbol{Q}_k \in \mathbb{R}^{d_x \times p}$$

  Calculate

$$\boldsymbol{g}_k = \sum_{i=1}^{p} \frac{f(\boldsymbol{x}_k + h_k \boldsymbol{q}_{k,i}) - f(\boldsymbol{x}_k)}{h_k}$$

  where $\boldsymbol{q}_{k,i}$ is the i'th column of $\boldsymbol{Q}_k$.
  Construct a subspace model

$$\hat{m}_k(\boldsymbol{\sigma}) = f(\boldsymbol{x}_k) + \boldsymbol{g}_k^\top \boldsymbol{\sigma} + \frac{1}{2}\boldsymbol{\sigma}^\top \boldsymbol{H}_k \boldsymbol{\sigma}$$

  Compute

$$\hat{\boldsymbol{\sigma}}_k \approx \arg\min\{\hat{m}_k(\boldsymbol{\sigma}) : \boldsymbol{\sigma} \in \mathbb{R}^p,\ \|\boldsymbol{\sigma}\|_2 \leq \Delta_k\}$$

  and calculate the step

$$\boldsymbol{\sigma}_k = \boldsymbol{Q}_k \hat{\boldsymbol{\sigma}}_k \in \mathbb{R}^{d_x}$$

  Evaluate $f(\boldsymbol{x}_k + \boldsymbol{\sigma}_k)$ and calculate ratio

$$\rho_k = \frac{f(\boldsymbol{x}_k) - f(\boldsymbol{x_k} + \boldsymbol{\sigma}_k)}{\hat{m}_k(\boldsymbol{0}) - \hat{m}_k(\hat{\boldsymbol{\sigma}}_k)}$$

  if $\rho_k \geq \eta_k$ **then**
    // SUCCESS ($b_k = 1$)
    Set $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \boldsymbol{\sigma}_k$ and $\Delta_{k+1} = \Delta_k$.
  else
    // FAILURE ($b_k = 0$)
    Set $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k$ and $\Delta_{k+1} = 0.8\Delta_k$.
  end if
end for

Our algorithm is heavily inspired by the STARS algorithm (Dzahini & Wild, 2022). The main idea of this algorithm is to calculate a ratio ρ, which is the relative difference between the model's prediction and the actual function. If this ratio falls below a certain acceptance threshold, then we will decrease the trust region area of $\boldsymbol{x}$. Otherwise, $\boldsymbol{x}$ is updated and we flag it as a successful iterator run. This algorithm is then treated as a "one-step iterator" in our full model, whose action hyper-parameters are trained through the NN and RL policies.

2.4. Neural network policy

2.4.1. Notation system

- d_x: dimension of $\boldsymbol{x}$.
- p: dimension of subspace ($p \leq d_x$).
- T: the window of number of iterations that we store the information in the $\boldsymbol{s}_t$.
- Input layer size: $T(d_x + p + 6) + 1$ where 1 is the bias/artificial neurons and 6 is the number of parameters at a specific state k included in the input information (difference parameter $h > 0$, trust region Δ, ratio ρ, acceptance threshold η, function value $f(\boldsymbol{x})$, and success b).
- Throughout the experiments settings, we use ReLU as the activation function. The NN has a simple structure with just 1 hidden simple layer.

2.4.2. NN architecture

For better interpretation purposes, we construct a simple NN structure with only one hidden layer with the input and hidden layers not fully connected. These two layers are connected such that each variables' values of one time step connect to the same neuron in the hidden layer. The hidden layer and the output layer are fully connected. The NN, in overall, chooses the action for the next state of our framework. The visual representation of our NN can be found in Figure 1.

2.5. Algorithm's flow

As aforementioned, we first generated a quadratic test problem and focused on dynamic policies for the RASTA randomized DFO algorithm. Through that process, we generated the initial action, which will act as the input of our NN. Here, the input of the neural network contains the variables' values at 5 consecutive time steps. This network will then return a new action and the next state, which will be fed into the RASTA randomized DFO algorithm again. This process is repeated multiple times, τ times in our case. The

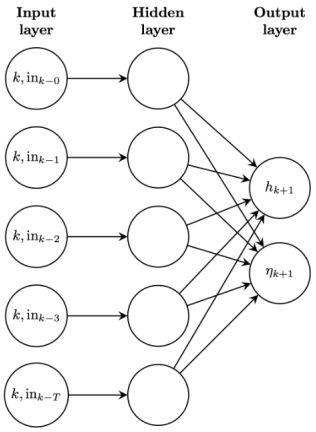

Figure 1. Neural network architecture. Here, k denotes the k-th iteration (or state) and $\text{in}_k = [h_k, \eta_k, f_k, \rho_k, \Delta_k, \boldsymbol{x}_k, b_k, \mathbf{g}_k]$.

NN's parameters will then be updated through a RL framework based on direct policy gradient. The objective of our framework is to maximize the reward function as a function of the network's parameters, $J(\mathbf{w})$, which is defined as the expected rewards for all trajectories on a given distribution. A detailed visual representation can be found in Figure 2.

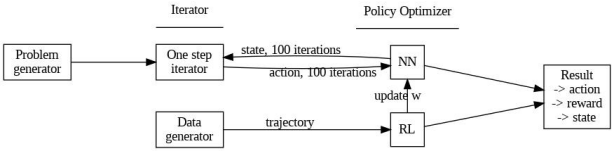

Figure 2. Algorithm's flow

2.6. Direct policy gradient

As mentioned briefly in Section 2.5, the reward function is defined as the expected rewards for all trajectories on a given distribution. In this section, we will go into more details regarding how to set up such function, as well as calculate its derivative with respect to the NN's parameters. This step is crucial as it helps us trace back and update the NN's weights for the next iteration.

Firstly, we denote $\tau = (\boldsymbol{s}_0, \boldsymbol{a}_0, \boldsymbol{s}_1, \boldsymbol{a}_1, \cdots, \boldsymbol{s}_T)$ to be the trajectories that are generated by the agent's interaction with the environment. Note that in our RL model, the parameters are the weights of the neural network, which are denoted as $\mathbf{w}$ above (a vector containing all weights of the network). We then define the distribution of the trajectories to be

$$p(\tau) = p(\boldsymbol{s}_0) \prod_{t=1}^{T_\tau} \pi_{\mathbf{w}}(\boldsymbol{a}_t|\boldsymbol{s}_t) p(\boldsymbol{s}_{t+1}|\boldsymbol{s}_t, \boldsymbol{a}_t)$$

We can write the distribution as such as in policy optimization, we care about learning an (explicit) parametric policy $\pi_{\mathbf{w}}$ with parameters $\mathbf{w}$. We then denote the expected reward for all trajectories on the given distribution to be

$$J(\mathbf{w}) = \mathbb{E}_{\tau \sim p(\tau)} \left[\sum_{t \geq 0} \gamma^t R(\boldsymbol{s}_t, \boldsymbol{a}_t) \right]$$

which can be estimated through sampling with N samples of trajectories

$$J(\mathbf{w}) = \frac{1}{N} \sum_i \sum_t \gamma^t R(\boldsymbol{s}_{i,t}, \boldsymbol{a}_{i,t})$$

Then, the objective is to maximize this expectation under the trajectory distribution

$$\mathbf{w}^* = \arg\max J(\mathbf{w})$$

To solve this, we need to figure out a way to compute the gradient of $J(\mathbf{w})$, which is $\nabla_{\mathbf{w}} J(\mathbf{w})$.

For a simpler notation, we introduce a compact notation,

$$r(\tau) = \sum_{t=1}^{T_\tau} \gamma^t R(\boldsymbol{s}_t, \boldsymbol{a}_t)$$

Additionally, the following is an obvious but useful identity regarding the derivative of $\log$ function,

$$p_{\mathbf{w}}(\tau) \nabla_{\mathbf{w}} \log p_{\mathbf{w}}(\tau) = p_{\mathbf{w}}(\tau) \frac{\nabla_{\mathbf{w}} p_{\mathbf{w}}(\tau)}{p_{\mathbf{w}}(\tau)} = \nabla_{\mathbf{w}} p_{\mathbf{w}}(\tau)$$

Recall the trajectory distribution, we can compute $\log p(\tau)$ as

$$\begin{aligned} \log p(\tau) &= \log \left[p(s_0) \prod_{t=1}^{T_\tau} \pi_{\mathbf{w}}(\boldsymbol{a}_t|\boldsymbol{s}_t) p(s_{t+1}|\boldsymbol{s}_t, \boldsymbol{a}_t) \right] \\ &= \log p(s_0) + \sum_{t=1}^{T_\tau} \log \left(\pi_{\mathbf{w}}(\boldsymbol{a}_t|\boldsymbol{s}_t) + p(s_{t+1}|\boldsymbol{s}_t, \boldsymbol{a}_t) \right) \end{aligned}$$

Now note that $\log p(s_0)$ and $p(s_{t+1}|\boldsymbol{s}_t, \boldsymbol{a}_t)$ do not depend on $\mathbf{w}$.

Aiming with these, $\nabla_{\mathbf{w}} J(\mathbf{w})$ can be rewritten as

$$\begin{aligned} \nabla_{\mathbf{w}} J(\mathbf{w}) &= \int \nabla_{\mathbf{w}} p_{\mathbf{w}}(\tau) r(\tau) d\tau \\ &= \int p_{\mathbf{w}}(\tau) \nabla_{\mathbf{w}} \log p_{\mathbf{w}}(\tau) r(\tau) d\tau \\ &= \mathbb{E}_{\tau \sim p_{\mathbf{w}}(\tau)} [\nabla_{\mathbf{w}} \log p_{\mathbf{w}}(\tau) r(\tau)] \\ &= \mathbb{E}_{\tau \sim p_{\mathbf{w}}(\tau)} \left[r(\tau) \left(\sum_{t=1}^{T_\tau} \nabla_{\mathbf{w}} \log \pi_{\mathbf{w}}(\boldsymbol{a}_t|\boldsymbol{s}_t) \right) \right] \\ &\approx \frac{1}{N_\tau} \sum_{i=1}^{N_\tau} \Big[\Big(\sum_{t=1}^{T_\tau} \nabla_{\mathbf{w}} \log \pi_{\mathbf{w}}(\boldsymbol{a}_{i,t}|\boldsymbol{s}_{i,t}) \Big) \\ &\qquad \Big(\sum_{t=1}^{T_\tau} R(\boldsymbol{s}_{i,t}, \boldsymbol{a}_{i,t}) \Big) \Big] \end{aligned}$$

where N_τ is the number of trajectories we're sampling on. The derivation composes of 2 parts: the first part is the sum of the network's outputs and the second part is the reward value at a specific state with a specific action. These two components are summed over all RASTA iterations.

We then update our weights by

$$\Delta \mathbf{w} = \beta \nabla_{\mathbf{w}} J(\mathbf{w})$$

where β is the learning rate for weight update. Thus,

$$\mathbf{w}_{k+1} = \mathbf{w}_k + \beta \nabla_{\mathbf{w}} J(\mathbf{w}_k)$$

2.7. Award functions

The final piece in our framework is the award functions. Due to the noise of the objective function, it is important to pick an appropriate rewards function as some are better at disambiguating the effects of the policy enacted. In this paper, we present with two different reward functions. The first reward function is only the inverse of the last functional value, proportional to the first functional value.

$$r(\tau) = \frac{f_{T_\tau}}{f_0} \tag{3}$$

The second one cares about the sum of the reward values, proportional to the first functional value.

$$r(\tau) = \frac{1}{f_0} \sum_{k=0}^{T_\tau} f_k \tag{4}$$

Experiments suggested that by using the second reward function, we can pick an optimal policy h for each test problem efficiently. Additionally, in one of our recent experiments, we showed that there exists an initial weight such that we obtain a fixed policy h and that h value is around 10^{-1}, which is a very reasonable one for every problem we tested here. More details regarding these results will be discussed in more details in Section 3.

3. Experimental results

As mentioned earlier in Section 2.7, we experimented with two different reward functions as stated in Equation (3) and Equation (4). The setup of the experiment is that we tried with different test problems and see how the reward functions behave with a fixed action h. As shown in Figure 3, the behavior of the first reward function (inverse of the last functional value) is not good enough as it cannot help the RL policy in picking an optimal policy. However, when we use the reward function as in Equation (4), we obtained the desired result. More specifically, for each test problem (corresponding to each line in Figure 3 and 4), we are able to choose an optimal action h that maximizes the reward values.

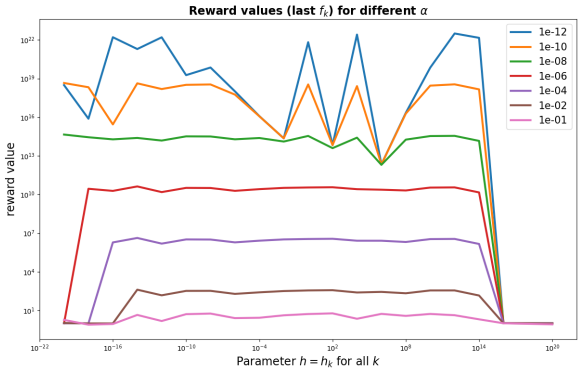

Figure 3. Reward function: inverse of the last functional value, shown in Equation (3)

We argue that since the reward function in Equation (3) is too simple and it solely depends on the last functional value, in case that f does not change much or initial f and last f are too similar, the reward function is not robust enough, leading to patterned, but nondeterministic, behaviors as shown in Figure 3.

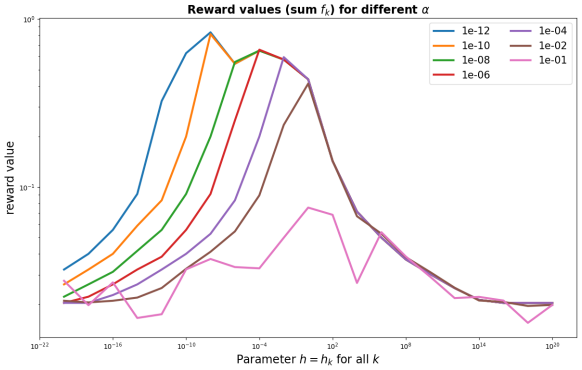

Figure 4. Reward function: sum of reward values, shown in Equation (4)

Figure 5 shows that our framework is able to choose the actions which in turn increase the reward. This way, the hyperparameters in the derivative-free optimization algorithm are adaptively chosen to solve the quadratic problem. Additionally, from the RL policy alone, the gradient of the network w.r.t the weights (shown in Figure 6) can come to an equilibrium as the number of iterations increases.

All the code used for these experiments can be found at this Github repository.

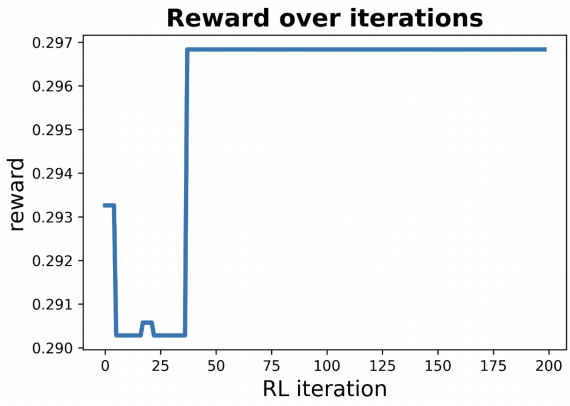

Figure 5. Rewards over iteration

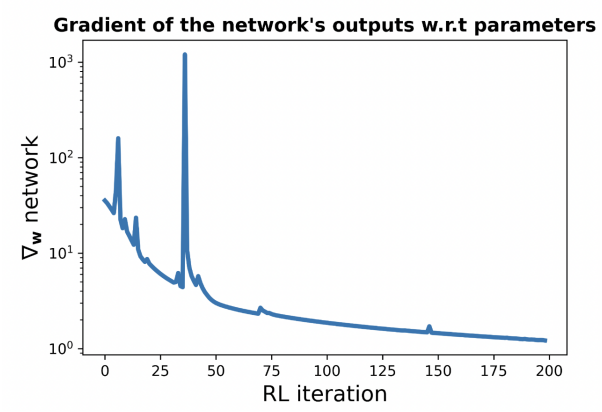

Figure 6. Rewards over iteration

4. Limitations & Future works

Having finished with the first stage of our project, we are working on adding more complexity to our problem. For now, we are using a fixed step size for our learning process but using adjusted step size might reduce the processing time and lead to better overall results. Currently, we only test our algorithm on quadratic problems since it is easy to interpret. In the long run, we expect our algorithm to work on more complicated ones and we are experimenting with Watson problems. Additionally, we want to implement more complicated structures for our neural network, for example adding more hidden layers and fully connected nets. Furthermore, we are interested in surveying more complex reward functions for our RL policy.

Lastly, we also want to try implementing parallelization in the testing process to speed up the performance. Currently, the running time of our pipeline is still quite long since each test problem is ran consecutively instead of in parallel. We are in the process of implementing a parallel pipeline so that we could do more extensive experiments more efficiently by utilizing the libEnsemble library. However, due to the time constraint, this work is only done partially and ran to some technical errors.

5. Acknowledgement

This work was made possible through the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, applied mathematics and SciDAC programs under Contract No. DE-AC02-05CH11231. We also would like to express gratitude to the staffs of Berkeley Lab, especially of AMCR division, for their numerous supports throughout.

References

Audet, C. and Hare, W. *Derivative-Free and Blackbox Optimization*. Springer Series in Operations Research and Financial Engineering. Springer International Publishing, 2018. ISBN 9783319886800. URL https://books.google.com/books?id=Csv5wQEACAAJ.

Cartis, C. and Roberts, L. Scalable subspace methods for derivative-free nonlinear least-squares optimization, 2021.

Dzahini, K. J. and Wild, S. M. Stochastic trust-region algorithm in random subspaces with convergence and expected complexity analyses. Technical Report 2207.06452, ArXiv, 2022. URL https://arxiv.org/abs/2207.06452.

Larson, J., Menickelly, M., and Wild, S. M. Derivative-free optimization methods. *Acta Numerica*, 28:287–404, May 2019. ISSN 1474-0508. doi:10.1017/s0962492919000060. URL http://dx.doi.org/10.1017/S0962492919000060.

Porcelli, M. and Toint, P. L. BFO, a trainable derivative-free brute force optimizer for nonlinear bound-constrained optimization and equilibrium computations with continuous and discrete variables. *ACM Transactions on Mathematical Software*, 44(1):1–25, 2017. doi:10.1145/3085592.

Shashaani, S., Hashemi, F. S., and Pasupathy, R. Astro-df: A class of adaptive sampling trust-region algorithms for derivative-free stochastic optimization. *SIAM Journal on Optimization*, 28(4):3145–3176, 2018.