MASTER CORO-SIP

SIGNAL AND IMAGE PROCESSING

2025 / 2026

Project Report

Presented by

Jayotsana SHARMA
Ryosuke MORI

On 24/06/2025

Reinforcement Learning Techniques for Solving Optimization Problems

Jury

Evaluators

Evaluator 1:	Clément Huneau	Professor, LS2N
Evaluator 2:	Sébastien Bourguignon	Professor, LS2N

Supervisors

Supervisor 1:	Diana Mateus Lamus	Professor, LS2N
Supervisor 2:	Nil Foix Colonir	PhD, LS2N

List of Figures

List of Tables

Contents

1 Introduction to Reinforcement Learning for Function Optimization

Reinforcement Learning (RL) has traditionally been used for sequential decision-making problems such as robotics, game playing, and control systems [2]. However, it is increasingly being applied to function optimization tasks especially in scenarios where the objective function is expensive to evaluate, non-differentiable, or defined over high-dimensional and continuous domains. Therefore, our goal is determine the global optimum of the such objective function $L(x)$:

$$x^* = \arg\max_x L(x) \quad \text{or} \quad x^* = \arg\min_x L(x).$$

In this context, the problem is framed as an episodic RL task, where the agent seeks to discover the global optimum of an unknown function by learning optimal actions (input state variables or coordinates) based on scalar feedback (function value at state input or reward). The RL-based optimization has been successfully applied in fields such as hyperparameter tuning, neural architecture search, industrial process control, and adaptive experimental design. It is especially useful when the search space is vast or non-convex, where traditional gradient-based or evolutionary optimization methods may struggle.

Li and Malik [3] are the first to emphasize the laborious process of designing traditional optimization algorithms that can be reformulated as a policy-learning problem. Their findings demonstrate that RL-based optimizers can outperform existing hand-engineered optimization methods.

Recently, Xu et al. [4] designed the REINFORCE-OPT algorithm to solve a wide range of optimization problems, including inverse problems and sparse reconstruction. They demonstrated that this algorithm outperforms other optimization methods, such as gradient descent, genetic algorithms, and particle swarm optimization. Its advantages include the ability to escape from locally optimal solutions and its robustness to the choice of initial values.

In this study, we focus on reproducing the REINFORCE-OPT algorithm for potential use in our inverse problem inspired by the promising results observed by Xu et al. [4]. Our goal is to demonstrate the efficacy of this method across various one-dimensional and two-dimensional functions known to be prone to local minima or maxima. Below are the contributions we made regarding this work.

1.1 Contributions

1. An updated code of REINFORCE-OPT is written in a proper class-based structure to facilitate the parameter variation study, which is available at `https://github.com/JayPanchariya/RLSparse.git`.

2. **Sections 3.1 and 3.2:** One and two dimensional functions are explored, which are known to be stuck at local minima. These are reproduced as given in the paper, along with new functions, especially Rosenbrock, double sinc function, and absolute value, which have not been explored in Xu et al. [4].

3. **Step 6:** Although REINFORCE-OPT has a specified stopping criteria based on fixed generation time, this study explored two new stopping criteria. The first criteria involves stopping when the policy gradients do not

change significantly for a specified threshold over 200 generations. The second criteria is based on the rewards, stopping when they do not change significantly.

4. **Section 3:** Given RL requires substantial computational resources and time, this study varied the parallel environments/batch size as L and the episodic length T to analyze the effects on computational time and results.

5. **Sections 2 and 1:** The actual code is written using the tf-agents library, which includes a lot of abstractions, such as not specifying the architecture of the policy network. In addition, a small piece of code has been added that operates independently of tf-agents, providing a more detailed understanding of how the system works.

1.2 Background of RL

Reinforcement Learning (RL) is a subfield of machine learning that models decision-making in dynamic environments through interactions between an agent and an environment, as shown in Figure 1. Unlike supervised learning, which learns from labeled data, RL is driven by feedback in the form of rewards. The agent learns to take actions that maximize cumulative reward over time, enabling adaptive behavior in uncertain or complex domains [1]. RL algorithms can be broadly

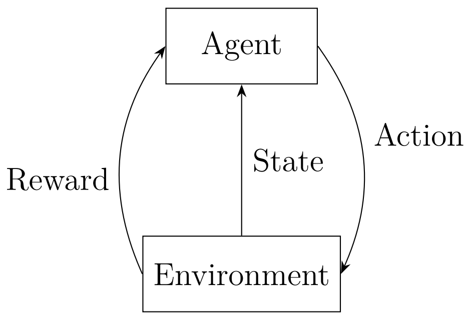


Figure 1: Schematic of Reinforcement Learning

categorized into **model-based and model-free approaches**. In function optimization, model-free methods are particularly valuable when the function model is unknown or intractable. Within model-free frameworks, **value-based methods such as Q-learning** estimate the expected returns of actions, while **policy-based methods** directly learn a mapping from states to actions. When it is integrated with deep learning, these methods form **Deep Reinforcement Learning (Deep RL)**, enabling the optimization of high-dimensional black-box functions [1]. Explaining approaches can be broadly categorized into two fundamental paradigms: **value-based methods** and **policy-based methods**. While both are effective in discrete environments, continuous action spaces introduce several limitations, particularly for value-based algorithms. Deep learning, through neural networks, provides the essential function approximation capability that enables these methods to scale and generalize to such complex settings.

1.2.1 Value-Based Methods

Value-based methods, such as *Q-learning*, focus on learning a **value function** typically the action-value function $Q(s, a)$—which estimates the expected cumulative reward of taking action a in state s, and following a certain policy thereafter. The optimal policy $\pi^*(s)$ is then derived by selecting the action that maximizes this value:

$$\pi^*(s) = \arg\max_a Q(s, a)$$

This approach is effective in discrete environments where the number of possible actions is finite and enumerable. However, it becomes impractical in **continuous state/action spaces**, where $a \in \mathbb{R}^n$, because the maximization operation $\arg\max_a Q(s, a)$ becomes computationally intractable. The agent would need to solve a nested optimization problem at every decision point, which is often inefficient or infeasible.

1.2.2 Policy-Based Methods

Policy-based methods, such as *REINFORCE*, overcome the limitations of value-based approaches in continuous or high-dimensional action spaces by directly parameterizing the policy $\pi_\theta(a|s)$. Here, θ denotes the parameters of the policy, typically the weights of a neural network. Rather than estimating value functions and deriving policies implicitly, policy-based methods aim to learn a policy that maximizes the expected cumulative return:

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta}[R(\tau)] \tag{1}$$

where τ represents a trajectory generated by the policy, and $R(\tau)$ is the cumulative reward along that trajectory.

The policy $\pi_\theta(a|s)$ defines a probability distribution over actions given the current state s. The specific form of this distribution depends on whether the action space is discrete or continuous.

1.2.3 Discrete Action Space

In discrete action spaces, the policy $\pi_\theta(a|s)$ is modeled as a **categorical distribution** over a finite set of k possible actions $\{a_1, a_2, \ldots, a_k\}$. The policy is parameterized by a neural network that outputs a probability vector $\mathbf{p}_\theta(s) = [p_1, p_2, \ldots, p_k]$, where:

$$p_i = \pi_\theta(a_i|s), \quad \sum_{i=1}^{k} p_i = 1, \quad p_i \geq 0 \tag{2}$$

These probabilities are typically computed using a softmax function applied to the output logits $\{z_1, z_2, \ldots, z_k\}$ of the neural network:

$$\pi_\theta(a_i|s) = \frac{\exp(z_i)}{\sum_{j=1}^{k} \exp(z_j)} \tag{3}$$

An action a_t is sampled from this distribution during interaction with the environment. The log-probability of the selected action is:

$$\log \pi_\theta(a_t|s_t) = \log p_{a_t} \tag{4}$$

The REINFORCE algorithm uses this log-probability to compute the policy gradient:

$$\nabla_\theta J(\theta) = \mathbb{E}_{a\sim\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s) \cdot R(a)\right] \tag{5}$$

This gradient adjusts the parameters θ to increase the probability of actions that result in higher rewards and decrease it for less effective actions over the time.

1.2.4 Continuous Action Space

Similarly, in continuous action spaces, the policy is typically modeled as a **multivariate Gaussian distribution**:

$$\pi_\theta(a|s) = \mathcal{N}(\mu_\theta(s), \sigma_\theta(s)) \tag{6}$$

Here, $\mu_\theta(s)$ and $\sigma_\theta(s)$ are the mean and standard deviation, both parameterized by neural networks. Actions $a \in \mathbb{R}^n$ are sampled from this distribution:

$$a \sim \mathcal{N}(\mu_\theta(s), \sigma_\theta(s)) \tag{7}$$

The policy gradient for continuous actions follows the same structure as in the discrete case, Eq. 5:

$$\nabla_\theta J(\theta) = \mathbb{E}_{a\sim\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s) \cdot R(a)\right] \tag{8}$$

By continuously adjusting the policy parameters based on observed returns, policy-based methods improve the likelihood of choosing actions from Neural Network that lead to higher long-term rewards. These methods are particularly effective in high-dimensional and continuous control tasks, where value-based methods may struggle due to the difficulty of action-value maximization.

2 Methodology used in REINFORCE-OPT

This section presents a brief overview of the REINFORCE-OPT framework, as described in [4]. It is a self-learning optimization approach that leverages a policy gradient method to explore and learn in environments with continuous state spaces, as explained in section method 1.2.2. The goal is to learn an optimal policy $\pi_\theta(a|s)$ that maximizes the expected return through repeated interaction with the environment, as shown in Fig. 2.

Figure 2 illustrates the core principles of deep reinforcement learning [1]. The **environment** defines the *search space* and provides the next steps for the agent. The **agent** interacts with the environment, while the **policy** guides the agent's actions, which can include making discrete choices, such as turning right or left, or taking continuous actions. Although the concepts of the agent, environment, and policy are straightforward, it is important to elaborate on them since they form the complete pipeline of the REINFORCE-OPT algorithm. However, it should be noted that in this study, we restrict our focus to the case of action-state:

- **Continuous state space:** States are represented by real-valued input vectors, enabling optimization over continuous domains.
- **Discrete action space:** The action set is limited to $\{-1, 0, 1\}$, representing discrete movement or update directions within the search space.

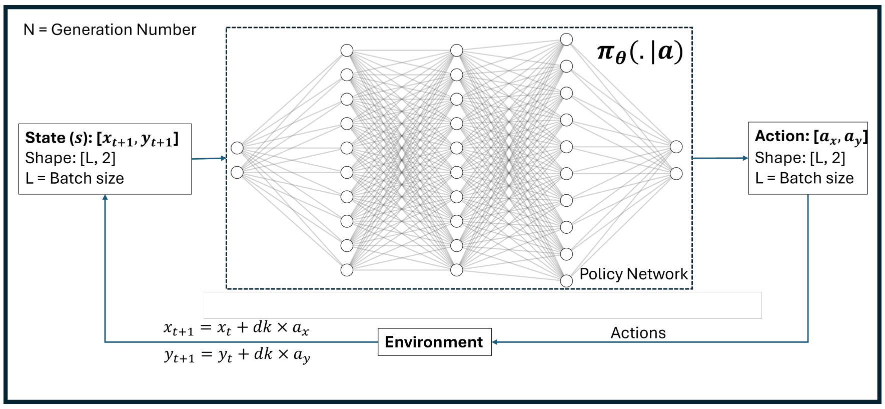


Figure 2: Schematic of the REINFORCE-OPT algorithm. At each time step t, the agent observes state s and samples actions a_x and a_y from the policy network $\pi_\theta(\cdot \mid s)$, where θ denotes the trainable network parameters. The step size Δk controls the magnitude of each update and R is reward obtained at s. The algorithm iterates for N generations or until a predefined maximum reward is achieved.

2.0.1 Environment

In our setup, the environment is implemented in the `environmentRL.py` and serves as the domain in which optimization takes place.

1. **Observation space:** The state $s_t \in \mathbb{R}^n$, representing the current point in the search space, is unbounded.
2. **Action space:** The agent selects discrete actions from $\{-1, 0, 1\}^n$ computed from policy network π_θ, as mentioned in subsection 2.0.2, which are later normalized to continuous direction vectors.
3. **Next step function:** The environment transitions to the next state based on the update rule:

$$s_{t+1} = s_t + \text{normalized\_action} \times \text{step\_size}$$

4. **Reward function:** The reward at each step is computed as the value of the objective function being optimized:

$$R_{t+1} = f(s_{t+1})$$

This environment generates a full trajectory under a given policy (initially random), forming an **episodic task** upto a finite horizon T define by user. A typical episode evolves as:

$$s_0, a_0, r_1, s_1, a_1, r_2, \ldots, s_T$$

2.0.2 Policy Agent π_θ

The policy network $\pi_\theta(a \mid s)$ takes the current environment state s_t and guides the agent to make the next choice based on the weights θ of the neural network. In this setup, the neural network consists of one input layer, three hidden layers with tanh activation functions, and an output layer that has six neurons, which allows for three choices for each two-dimensional input, as shown in Fig. 2. The weights are randomly initialized using a normal distribution. The policy network $\pi_\theta(a \mid s)$ is parameterized by as follows:

$$\theta = \left\{W^{(l)},\, b^{(l)}\right\}_{l=1}^{4}$$

where $W^{(l)}$, $b^{(l)}$ denote the weight matrix and bias vector of the l^{th} layer, respectively and maps a continuous state $s \in \mathbb{R}^d$ to a 6-dimensional output whose first three entries control the x-direction action and whose last three control the y-direction action. Concretely:

$$\begin{aligned} h^{(0)} &= s, \\ h^{(l)} &= \tanh\Big(W^{(l)}\, h^{(l-1)} + b^{(l)}\Big), \quad l = 1, 2, 3, \\ z &= W^{(4)}\, h^{(3)} + b^{(4)} = \begin{pmatrix} z_x \\ z_y \end{pmatrix}, \quad z_x, z_y \in \mathbb{R}^3. \end{aligned}$$

We then apply a separate softmax over each 3-vector:

$$\pi_\theta(a_x = i \mid s) = \frac{\exp(z_{x,i})}{\sum_{j=1}^{3} \exp(z_{x,j})}, \quad \pi_\theta(a_y = k \mid s) = \frac{\exp(z_{y,k})}{\sum_{j=1}^{3} \exp(z_{y,j})},$$

and sample the two-dimensional action $a = (a_x, a_y)$ accordingly.

2.1 Algorithm Procedure

Below is the detailed, step-by-step procedure for reproducing the REINFORCE-OPT algorithm.

1. **Define reward:** Let $f(x)$ be the objective as reward function.
2. **Initialize:**
   - State: $\mathbf{x}_0 \leftarrow$ `self.x0_reinforce`
   - Policy network: instantiate π_θ as in Section 2.0.2.
3. **Collect trajectory.** For $t = 0, 1, 2, \ldots$ until termination T:
   (a) *Action selection:* Sample from $a_t \sim \pi_\theta(\cdot \mid \mathbf{s}_t)$
   (b) *Environment step.* Execute a_t; observe
   $$\begin{aligned} &\mathbf{s}_{t+1} \text{ via the transition function (cf. Fig. 2)}, \\ &r_t = f(\mathbf{s}_t) \end{aligned}$$
   (c) *Store transition.* Record $\{(\mathbf{s}_t^{(j)}, a_t^{(j)}, r_t^{(j)})\}_{j=1}^{T}$
4. **Parallel environments (L):** Instantiate L independent copies of the environment to sample trajectories concurrently. At each time step t, collect the batch of states
   $$\{\mathbf{s}_t^{(i)}\}_{i=1}^{L},$$
   feed them into the policy network π_θ as a batch of size L, and sample the corresponding actions
   $$\{a_t^{(i)}\}_{i=1}^{L}.$$
   Record $\{\{(\mathbf{s}_t^{(j)}, a_t^{(j)}, r_t^{(j)})_{j=1}^{T}\}^{(i)}\}_{i=1}^{L}$ for L batches and T trajectory.

5. **Policy update:** After a full trajectory T with batch L, compute the policy gradient and update θ via the REINFORCE rule:

   (i) Compute gradient:

$$\hat{\nabla}_\theta J_T(\theta) = \frac{1}{L}\sum_{t=0}^{L-1}\left[R(h_T)\sum_{t=0}^{T-1}\nabla_\theta \log \pi_\theta(a_t \mid s_t)\right]$$

   where $R(h_T)$ is summation over T

$$R(h_T) = \frac{1}{T}\sum_{t=0}^{T-1} r_t$$

   (ii) Update Rule:

$$\theta_{n+1} = \theta_n + \alpha_n\left(\hat{\nabla}_\theta J_T(\theta_n) - 2\beta\theta_n\right)$$

   where β is regularization coefficient and α is learning rate.

6. **Repeat.** Return to Step 3 with the updated policy up to N generation.

   (i) Select the trajectories with the highest rewards:

$$r^* \;=\; \max_{j\in\text{batch}} R(\tau_j)\,,$$

   (ii) Or monitor the ℓ_2–norm of the policy gradient,

$$\left\|\nabla_\theta J(\theta)\right\|_2 \;=\; \sqrt{\sum_i\left(\frac{\partial J(\theta)}{\partial\theta_i}\right)^2}\,,$$

   and stop when it falls below a chosen threshold.

3 Experiments

We evaluate the REINFORCE-OPT algorithm as defined in section 2.1 on a diverse set of benchmark functions as shown below as one-dimensional and two-dimensional test functions. These functions feature non-differentiable regions and multiple local extrema, challenging standard gradient-based methods.

3.1 One-Dimensional Test Functions

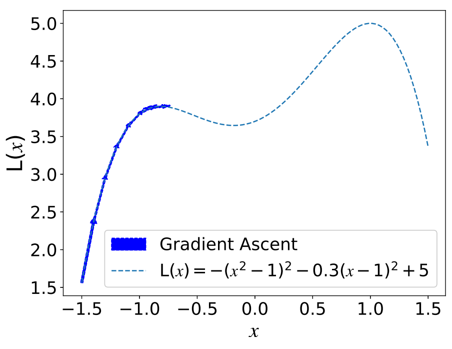
We consider two parameterized test functions problems in $\mathbb{R}$:

Problem 1: $f(x) \;=\; (x^2-1)^2 \;+\; \lambda\,(x-1)^2 \;+\; 5 \quad \lambda \geq 0$. This quartic function exhibits multiple local minima whose locations depend on λ.

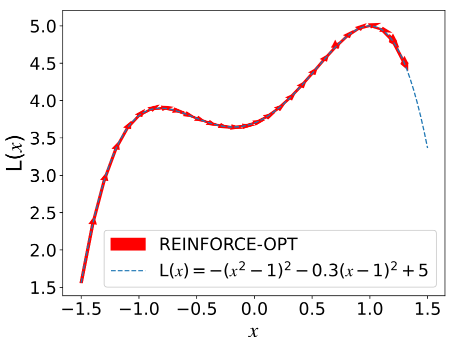
Problem 2: $f(x) \;=\; \sin(3x) \;+\; \lambda\,x^2, \quad \lambda \geq 0$. A highly non-convex, oscillatory landscape regularized by the quadratic term.

Table 1: Problem 1 hyperparameter for RL

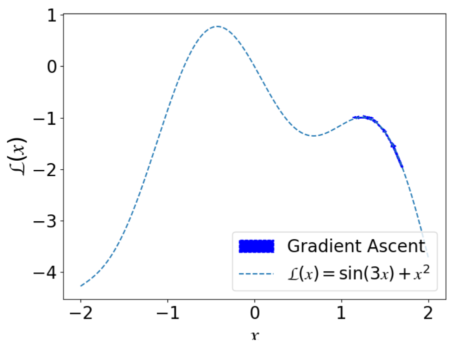
REINFORCE Optimization		**Policy Network**	
Parameter	**Value**	**Parameter**	**Value**
Number of generations	3,500	Hidden layers	3
Initial state $\mathbf{s}_0$	-1.5	Units per layer	16
Step size dk	0.1	Activation	tanh
Trajectory length T	35	Learning rate α	1×10^{-5}
Parallel environments	10	Optimizer	SGD
Batch size L	$10 \times 6 = 60$	Regularization coefficient β	0.15



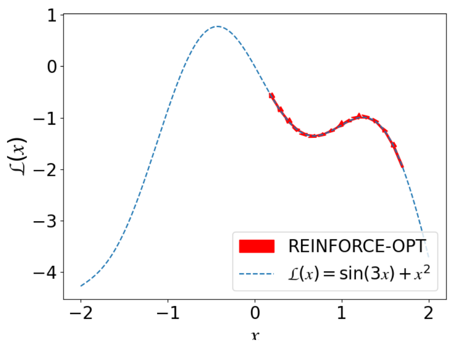
(a)



(b)



(c)



(d)

Figure 3: Comparison of trajectories for one-dimensional functions. Figures (a) and (b) correspond to Problem 1, showcasing the trajectories obtained via (a) gradient-based optimization (blue) and (b) RL, respectively (red). Figures (c) and (d) correspond to Problem 2, where the same methods (c) gradient-based (blue) and (d) RL (red) are applied to a more complex sinusoidal function.

Table 2: Problem 2 hyperparameter for RL

REINFORCE Optimization		**Policy Network**	
Parameter	**Value**	**Parameter**	**Value**
Number of generations	4,500	Hidden layers	3
Initial state $\mathbf{s}_0$	1.7	Units per layer	16
Step size dk	0.1	Activation	tanh
Trajectory length T	25	Learning rate α	1×10^{-5}
Parallel environments	10	Optimizer	SGD
Batch size L	$10 \times 6 = 60$	Regularization coefficient β	0.0001

For both Problems 1 and 2, the hyperparameter settings for the algorithm mentioned in Tables 1 and 2 are provided. The results are illustrated in Figs. 3 (a, b, c, d) for both problems. These results show that the gradient-based optimization method becomes stuck at local maxima, while the RL based optimization successfully escapes local maxima and reaches the global maximum for both functions. However, it is important to note that in Fig. 2 (d), we were unable to reach the global minimum. This was due to the requirement of running more generations, but after 4500 generations, the process was terminated by a segmentation fault because of limited memory on the computer. Nonetheless, it is undeniable that the RL approach was able to escape local maxima; it simply requires more generations, or, in other words, additional maximum rewards.

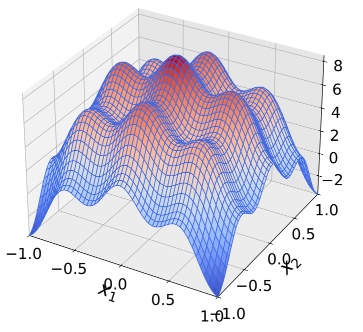
3.2 Two-Dimensional Test Functions

We further test on three classic functions problems in $\mathbb{R}^2$:

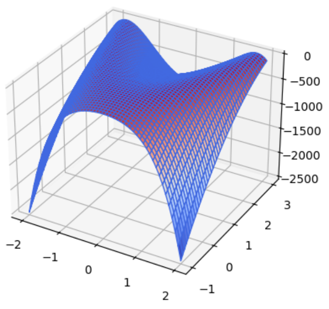
Problem 3: $f(x_1, x_2) = 8 + cos(10x_1) + cos(10x_2) + 5(x_1^2 + x_2^2))$; A highly multimodal function combining periodic and quadratic components.

Problem 4: **(Rosenbrock)** $f(x_1, x_2) = -((1 - x_1)^2 + 100(x_2 - x_1^2)^2)$. The classic "banana" function with a narrow curved valley and a single global maximum.

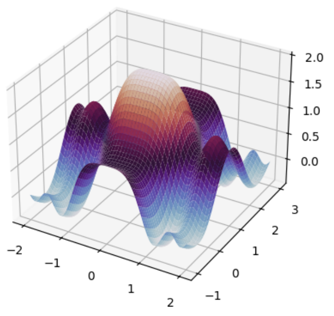
Problem 5: **(Double Sinc)** $f(x_1, x_2) = \operatorname{sinc}\big(x_1^2\big) + \operatorname{sinc}\big(x_2 - 1\big)$. A non-convex surface with many local maxima and minima.



(a)



(b)



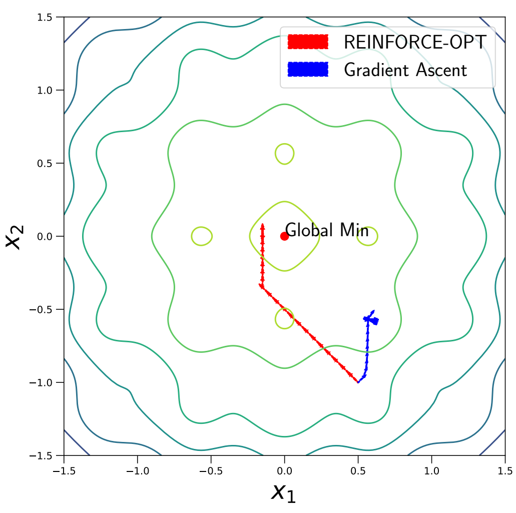
(c)

Figure 4: Three 2D test functions used in our optimization study as problems: (a) Multimodal function combining periodic and quadratic terms (Problem 3) (b) Rosenbrock function, illustrating a narrow curved valley (Problem 4), and (c) Double Sinc function, featuring multiple oscillatory extrema (Problem 5).

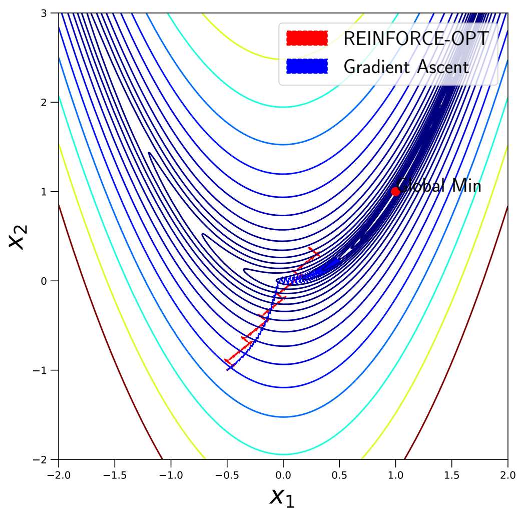
A similar procedure is used for optimizing 2D functions as Problem 3, 4 and 5 with RL, as defined in section 2.1. The functions are illustrated in Fig 4, and the corresponding RL-OPT hyperparameter settings are detailed in Tables 3, 4, and 5. For problems 3 and 4, contour plots are displayed in Fig 5(b). Both problems utilize a generation number of 16000 but start from different initial states. It can be observed that the gradient optimizer is stuck in both Figures 5(b) (a) and (b), while the RL optimization shows a tendency to approach the global minimum. This comparison clearly demonstrates that the RL method is effective in finding global minima by exploring various trajectories and maximizing the log gradient of the policy network.

Table 3: Problem 3 hyperparameter for RL

REINFORCE Optimization		**Policy Network**	
Parameter	**Value**	**Parameter**	**Value**
Number of generations	16,000	Hidden layers	3
Initial state $\mathbf{s}_0$	$[0.5, -1]$	Units per layer	16
Step size dk	0.05	Activation	tanh
Trajectory length T	30	Learning rate α	1×10^{-5}
Parallel environments	10	Optimizer	SGD
Batch size L	$10 \times 6 = 60$	Regularization coefficient β	0.0001



(a)



(b)

Figure 5: Comparison of contour plots trajectories for two-dimensional functions. Contour plots (a) and (b) correspond to Problem 3 and 4, showcasing the trajectories obtained via (a) gradient optimization (blue) and (b) RL, respectively (red) after a N generations.

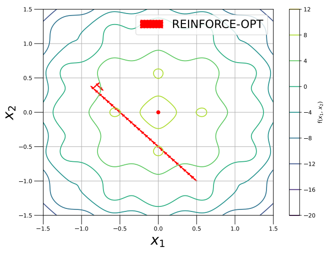
Table 4: Problem 4 hyperparameter for RL

REINFORCE Optimization		**Policy Network**	
Parameter	**Value**	**Parameter**	**Value**
Number of generations	16,000	Hidden layers	3
Initial state $\mathbf{s}_0$	$[0.5, -1]$	Units per layer	16
Step size dk	0.05	Activation	tanh
Trajectory length T	30	Learning rate α	1×10^{-5}
Parallel environments	10	Optimizer	SGD
Batch size L	$10 \times 6 = 60$	Regularization coefficient β	0.0001

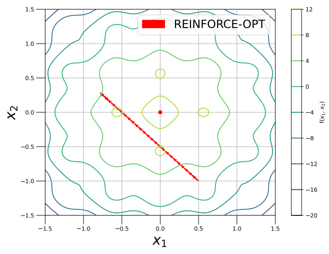
Table 5: Problem 5 hyperparameter for RL

REINFORCE Optimization		**Policy Network**	
Parameter	**Value**	**Parameter**	**Value**
Number of generations	8,000	Hidden layers	3
Initial state $\mathbf{s}_0$	$[0, -0.5]$	Units per layer	16
Step size dk	0.05	Activation	tanh
Trajectory length T	30	Learning rate α	1×10^{-5}
Parallel environments	10	Optimizer	SGD
Batch size L	$10 \times 6 = 60$	Regularization coefficient β	0.0001

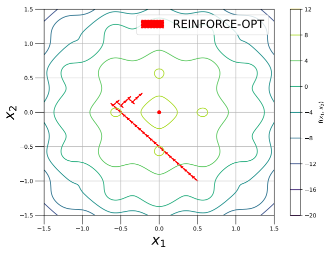
During training of policy network the behavior of trajectories from RL-OPT for Problems 3, 4 and 5 are shown in Figures 6, 7 and 8. Here we can observe that the steps are moving towards the optima as the generation number is increasing. It shows that the policy network parameters or weights are learning as it gets more inputs and tries to increase the cumulative reward for each trajectory. Concretely, after each generation we record the best step the state with the highest reward value encountered. We then use this information to compute the REINFORCE gradient, as mentioned in section 2.1 with step 5. Importantly, by continuously sampling new trajectories and always comparing against the global best seen so far rather than simply following a local gradient—the algorithm avoids becoming permanently trapped in shallow local extrema. Instead, it persistently explores for higher reward states, allowing the policy network to correct course when a better path is found.



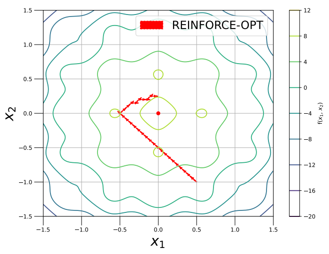
(a) Plot 300



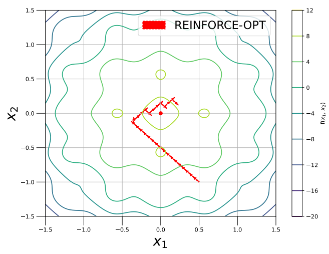
(b) Plot 1100



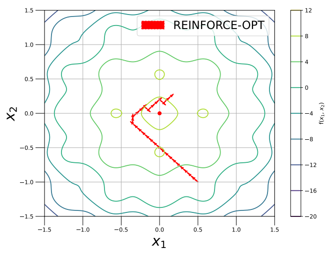
(c) Plot 1600



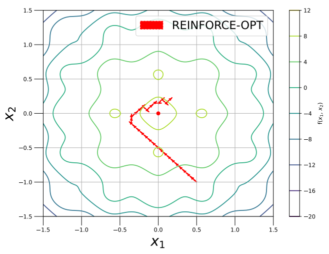
(d) Plot 2300



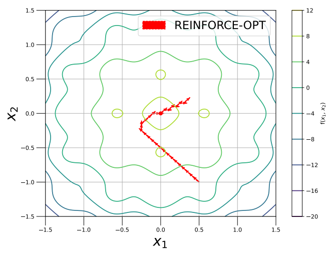
(e) Plot 4600



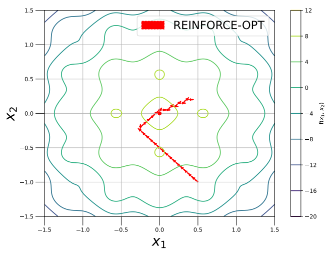
(f) Plot 5000



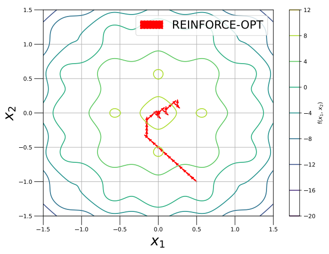
(g) Plot 5900



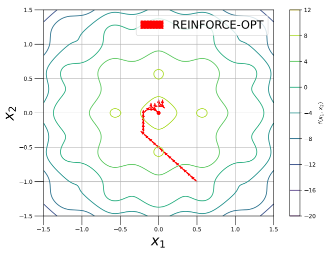
(h) Plot 7100



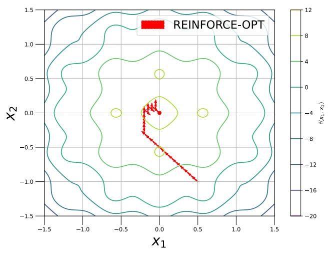
(i) Plot 8000



(j) Plot 14300



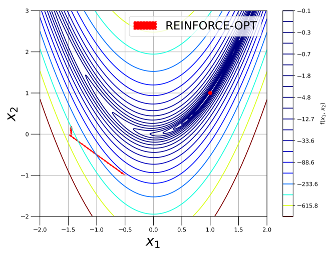
(k) Plot 15300



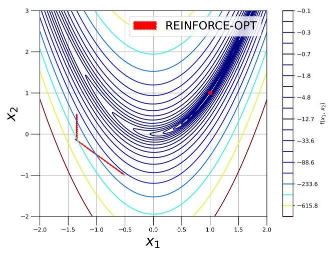
(l) Plot 15400

Figure 6: Intermediate policy-network trajectories overlaid on the contour plot of Problem 3's objective function during training.

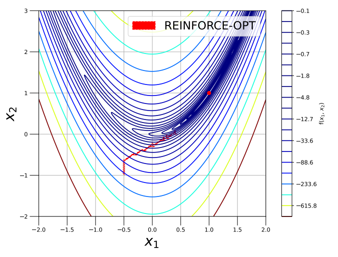
In Figure 6, we can see that the global maximum for the function occurs at the state (0, 0), where the highest value of objective function is 10. The trajectory is moving toward the maximum, and we achieve this maximum reward of 10 at the optimal step. After this step, no matter how many iterations are performed, the trajectory continues to move, as the policy does not stop in its pursuit of other potential rewards. However, the best reward remains constant at 10, with no other step yielding a higher reward. We halted the learning process after 16,000 iterations.



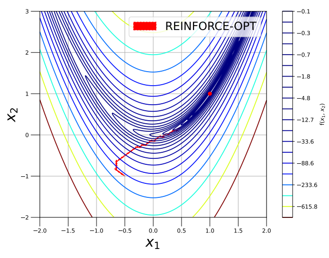
(a) Plot 100



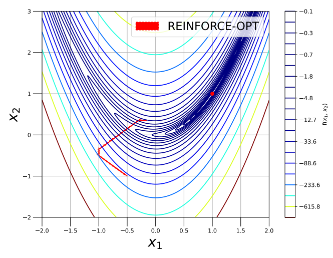
(b) Plot 1000



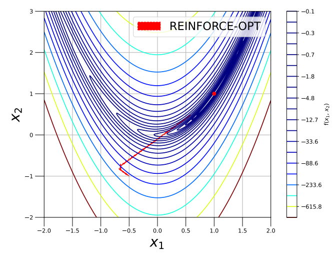
(c) Plot 2900



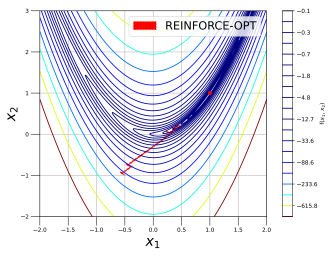
(d) Plot 3000



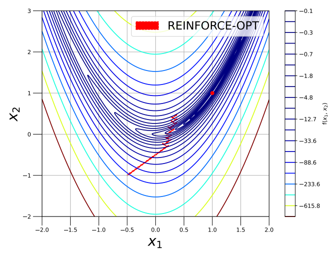
(e) Plot 3400



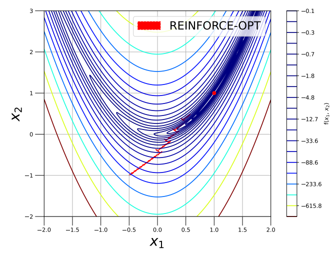
(f) Plot 4300



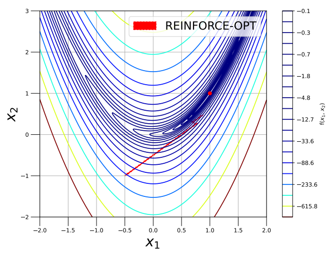
(g) Plot 5600



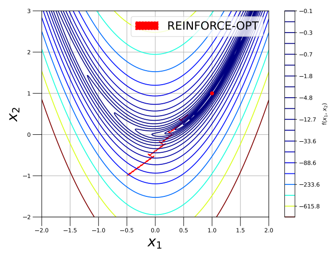
(h) Plot 7900



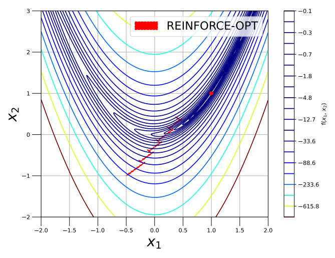
(i) Plot 8100



(j) Plot 10100



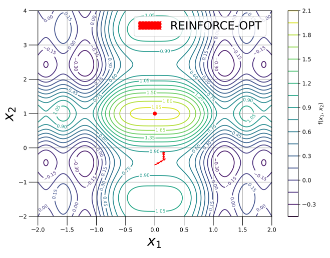
(k) Plot 15600



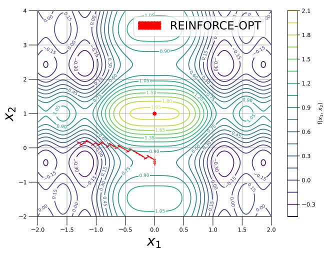
(l) Plot 16000

Figure 7: Intermediate policy-network trajectories overlaid on the contour plot of Problem 4's objective function during training.

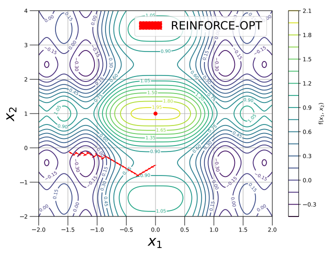
In Figure 7 for Problem 4, we can observe the agent's learning progression over 16,000 generations as its trajectory moves towards the global minimum. However, by the 16,000th generation, we did not achieve the best step or the optimal reward. Ideally, the best step and the corresponding maximum reward should occur at the point (1,1), where the reward is 0. Up until generation 16,000, the highest reward obtained was approximately -0.17, indicating that many more iterations may be necessary.



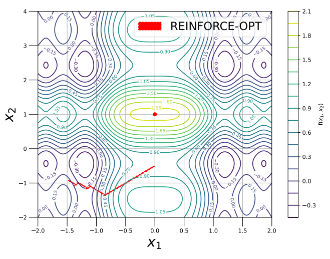
(a) Plot 0



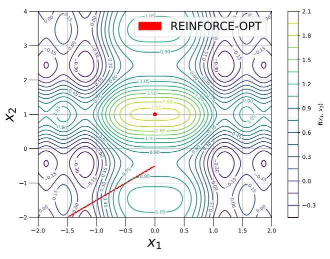
(b) Plot 100



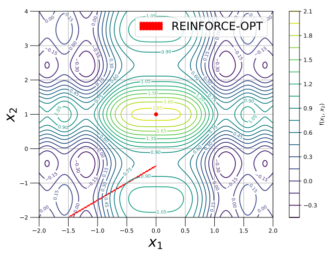
(c) Plot 200



(d) Plot 300



(e) Plot 400



(f) Plot 8000

Figure 8: Intermediate policy-network trajectories overlaid on the contour plot of Problem 5's objective function during training.

In Figure 8, we used RL-Opt to search for the global optimum by trying many different paths, as you can see in the early plots. We only ran it for 8,000 generations because of time limits. After about generation 400, the best reward stopped improving—even though the agent kept moving around which means it got stuck in a good local maximum. To find the true global maximum, it would need several thousand more iterations.

4 Conclusion

In this study, we used RL-OPT a policy gradient reinforcement learning algorithm to optimize both one-dimensional and two-dimensional as a loss functions, demonstrating its ability to escape local minima that often trap conventional methods. Over a sequence of N generations, the policy network incrementally learns a mapping from the current state to the next action that maximizes cumulative reward, where reward is defined directly in terms of the objective function value. After sufficient training typically several thousand generations—the optimized policy consistently produces near-optimal trajectories and yields highly optimized solutions even for highly non-convex test functions on which state-of-the-art algorithms stagnate. However, It is observed that training RL-OPT on a 2017 Mac Pro without GPU acceleration presented significant challenges. Optimizing a simple one-dimensional test function took approximately 7 hours of wall-clock time, while extending the optimization to a two-dimensional Rosenbrock-style function increased the runtime to around 10 hours, whereas gradient based methods take typically few seconds.

To optimize computation time, the hyperparameters in the algorithm were carefully tuned. Reducing the number of parallel environments L to 1 resulted in a decrease in computation time to 2.3 seconds per iteration. In contrast, using a higher number of parallel environments, such as ten, often yielded better and more optimal results, but at the cost of increased computation time, averaging

4.58 seconds per iteration, along with a higher memory burden.

Additionally, changes in episode length T and step size dk had a significant impact on computation time. Decreasing the episode length and using larger step sizes reduced computation time; however, this approach made it more difficult to reliably search for optimal solutions.

A critical issue we encountered is that standard policy-gradient implementations rely on a fixed number of generations as the sole stopping criteria. In practice, this "train-for-N generations and then stop" approach offers no guarantee that the learned policy has converged to a near-optimal solution, and may waste resources once further improvements become negligible or plateau. To address this, we evaluated two early-stopping criteria:

1. **Weight-norm criteria**. We monitor the change in the ℓ_2 norm of the policy network's weights across successive generations. If the norm does not change by more than a small threshold ϵ_w (e.g., 10^{-6}) for 200 consecutive generations, we interpret this as convergence in policy parameters and terminate training.

2. **Reward-improvement criteria**. We track the best achieved reward (i.e., highest objective value) and stop if this value fails to increase by at least ϵ_r (e.g., 10^{-3}) over 200 consecutive generations.

Using the above stopping criteria, we observed a risk of premature convergence if the global optimum is far from the last best step. The policy's exploratory nature might decay before reaching the true global maximum. Therefore, sometimes our experiments are truncated early.

5 Future works

1. In this work, we studied the RL-OPT method for a limited number of functions, specifically up to two-dimensional functions. It would be interesting to explore more challenging functions with higher dimensions. Additionally, we should investigate variations in the number of steps, step sizes, and batch sizes for a fixed number of generations. Exploring different configurations of the policy network, such as varying the number of layers and types of activation functions, could also be beneficial.

2. This study noted that the stopping criteria for the RL-OPT algorithm was not implemented. Although we attempted to implement two stopping criteria, we did not succeed; testing these could yield valuable insights. However, it is straightforward to apply a stopping criteria based on achieving a mean squared error of zero for inverse problems.

3. The implementation was carried out using TensorFlow and the tf-agents library. However, we faced challenges using the GPU in a parallel environment. This limitation could be addressed by utilizing the PyTorch library, which allows for training the policy network on a GPU.

4. Additionally, we confined our study to discrete action spaces using log probabilities, but it would be advantageous to incorporate continuous actions modeled with a normal distribution.

References

[1] Barto Andrew and Sutton Richard S. Reinforcement learning: an introduction. 2018.

[2] Peter Henderson, Joshua Romoff, and Joelle Pineau. Where did my optimum go?: An empirical analysis of gradient descent optimization in policy gradient methods, 2018. URL https://arxiv.org/abs/1810.02525.

[3] Ke Li and Jitendra Malik. Learning to optimize, 2016. URL https://arxiv.org/abs/1606.01885.

[4] Chen Xu, Yun-Bin Zhao, Zhipeng Lu, and Ye Zhang. Reinforcement-learning-based algorithms for optimization problems and applications to inverse problems, 2025. URL https://arxiv.org/abs/2310.06711.

A Code for `policyNet.py`

```python

import tensorflow as tf
import tensorflow_probability as tfp
tfd = tfp.distributions

class PolicyNet(tf.keras.Model):
    def __init__(self):
        super().__init__()
        self.d1 = tf.keras.layers.Dense(32, activation='tanh')
        self.d2 = tf.keras.layers.Dense(32, activation='tanh')
        self.logits_x = tf.keras.layers.Dense(3)  # for action x
        self.logits_y = tf.keras.layers.Dense(3)  # for action y

    def call(self, state):
        x = self.d1(state)
        x = self.d2(x)
        logits_x = self.logits_x(x)
        logits_y = self.logits_y(x)
        return tfd.Categorical(logits=logits_x),
    ↪ tfd.Categorical(logits=logits_y)
```

Listing 1: The full source of `policyNetework`

B Code for `RLOPT.py`

```python
from tf_agents.environments.parallel_py_environment import
    ↪ ParallelPyEnvironment
import matplotlib.pyplot as plt
from matplotlib import cm
from mpl_toolkits.mplot3d import axes3d
from matplotlib.axes import Axes as ax
```

```
import numpy as np
import tensorflow as tf
import os,gc
import time
import environmentRL as ENV
from typing import Tuple
from tensorflow.keras import layers
import tensorflow_probability as tfp
import policyNet
from tf_agents.system import multiprocessing


class MyTFPolicy(tf_policy.TFPolicy):
    def __init__(self, time_step_spec, action_spec, policy_net):
        self._policy_net = policy_net
        self._action_spec = action_spec
        super().__init__(time_step_spec, action_spec)

    def _distribution(self, time_step, policy_state):
        dist_x, dist_y = self._policy_net(time_step.observation)
        return policy_step.PolicyStep(distribution=(dist_x, dist_y),
    ↪ state=policy_state)

    def _action(self, time_step, policy_state, seed=None):
        dist_x, dist_y = self._policy_net(time_step.observation)
        action_x = dist_x.sample(seed=seed)
        action_y = dist_y.sample(seed=seed)
        actions = tf.stack([action_x, action_y], axis=-1)
        return policy_step.PolicyStep(action=actions,
    ↪ state=policy_state, info={})


def makeEnv(EnvClass, env_num=2):
    parallel_env =
    ↪ ParallelPyEnvironment(env_constructors=[EnvClass]*env_num,
                                          start_serially=True,
    ↪ blocking=False,flatten=False)
    return parallel_env


def run_batch_episode(env, policy, max_steps=200):
    time_step = env.reset()
    state = time_step.observation  # shape: [batch, ]

    batch_log_probs = []
    batch_rewards   = []
    for t in range(max_steps):
        dist_x, dist_y = policy(state)
        action_x = dist_x.sample()
        action_y = dist_y.sample()
        log_prob = dist_x.log_prob(action_x) +
    ↪ dist_y.log_prob(action_y)
```

```
        action = tf.stack([tf.cast(action_x,
    ↪ tf.float32),tf.cast(action_y, tf.float32)], axis=-1)    # now
    ↪ shape [batch, 2]
        time_step = env.step(action)
        batch_rewards.append(time_step.reward)
        batch_log_probs.append(log_prob)
        state = time_step.observation


    rewards = tf.stack(batch_rewards,   axis=0)  # [T, batch]
    log_probs = tf.stack(batch_log_probs, axis=0)  # [T, batch]
    returns = tf.reduce_sum(rewards, axis=0)      # shape [batch]

    return returns, log_probs




@tf.function
def train_step(batch_env, policy, optimizer):
    with tf.GradientTape() as tape:
        rewards, log_probs = run_batch_episode(batch_env, policy,
    ↪ max_steps=60)
        variables_to_train=policy.trainable_variables
        m = rewards.shape[0]
        regu_term = tf.reduce_sum(variables_to_train[0]**2)
        num = len(variables_to_train) #number of vectors in
    ↪ variables_to_train
        for i in range(1,num):
            regu_term += tf.reduce_sum(variables_to_train[i]**2)
        loss = -tf.reduce_sum(tf.reduce_sum(log_probs, axis=0)*
    ↪ rewards)/m+ 0.15*regu_term
    grads = tape.gradient(loss, policy.trainable_variables)
    optimizer.apply_gradients(zip(grads, policy.trainable_variables))

    return tf.reduce_sum(rewards), loss


if __name__ == '__main__':
    optimizer = tf.keras.optimizers.Adam(learning_rate=0.001)
    batch_size = 32
    batch_env =
    ↪ tf_agents.environments.BatchedPyEnvironment([ENV.Env()]*batch_size,
    ↪ multithreading= False)
    tf_env = tf_py_environment.TFPyEnvironment(batch_env )
    # tf_env=makeEnv(ENV.Env, env_num=40)
    policy = policyNet.PolicyNet()
    for iteration in range(10000):
        returns, loss=train_step(tf_env , policy, optimizer)
        if iteration % 2 == 0:
            print(f"Episode␣{iteration}:␣return␣=␣{returns:.2f},␣loss␣
    ↪ =␣{loss:.4f}")
```

Listing 2: The full source of `RLOPT.py`