Switching policies for solving inverse problems

Tim Bakker[*]
University of Amsterdam
t.b.bakker@uva.nl

Fabio Valerio Massoli
Qualcomm AI Research

Thomas Hehn
Qualcomm AI Research

Tribhuvanesh Orekondy
Qualcomm AI Research

Arash Behboodi
Qualcomm AI Research

Abstract

In recent years, inverse problems for black-box simulators have enjoyed increased focus of the machine learning community due to their prevalence in science and engineering domains. Such simulators describe a forward process $f : (\boldsymbol{\psi}, \boldsymbol{x}) \to \boldsymbol{y}$. Here the intent is to optimise simulator parameters $\boldsymbol{\psi}$ to minimise some observation loss on $\boldsymbol{y}$, under some input distribution on $\boldsymbol{x}$. Optimisation of such objectives is often challenging, since it is not trivial to estimate simulator gradients accurately. In settings where multiple related inverse problems need to be solved simultaneously, from-scratch/ab-initio optimisation of each may be infeasible if the forward model is expensive to evaluate. In this paper, we propose a novel method for solving such families of inverse problems with reinforcement learning. We train a policy to guide the optimisation by selecting between gradients estimated numerically from the simulator and gradients estimated from a pre-trained surrogate model. After training the surrogate and the policy, downstream inverse problem optimisations require 10%-70% fewer simulator evaluations. Moreover, the policy does successful optimisations on functions where using just simulator gradient estimates fails.

1 Introduction and related work

In many scientific and engineering fields, deducing unknown properties or parameters of a system from observed data is a recurring problem [9]. These problem settings are known as *inverse problems*. They pervade fields such as particle physics [35], wireless applications [27], medical imaging [28; 3], molecular dynamics [18] and design [32], and they even find applications in sustainability [8]. Traditionally, these scenarios involve a forward simulator model $f_s : (\boldsymbol{\psi}, \boldsymbol{x}) \to \boldsymbol{y}$ that maps continuous simulator parameters $\boldsymbol{\psi}$ and input data $\boldsymbol{x}$ into observations $\boldsymbol{y}$. For instance, in the context of particle physics, f_s may simulate the detection of muons $\boldsymbol{y}$, given properties of particles $\boldsymbol{x}$ and specific detector settings $\boldsymbol{\psi}$. These simulations form a critical component in efficiently designing real-world experiments, potentially expediting experimental research. Nonetheless, the high computational demands of simulators often requires optimisation with a minimal number of simulator calls. Gradient-based optimisation is effective when dealing with (approximately) differentiable simulators [36; 13; 11; 15; 12; 23], but a substantial portion of applications involves non-differentiable or completely black-box simulators [21; 9; 26]. This paper focuses on the subset of inverse problems where the goal is to optimise the parameters $\boldsymbol{\psi}$ of some black-box simulator f_s under some loss function $\mathcal{L}(\boldsymbol{y})$. We will consider the problem of minimising muon detection events as a running example [34; 6].

If simulators are not directly differentiable, optimisation can still be performed using numerical differentiation [2; 33], evolutionary strategies [5; 22], or Bayesian Optimisation [25; 10]. Another viable strategy is to leverage a surrogate model to perform gradient-based optimisation. The gradients

[*]Work done during internship at Qualcomm AI research

37th Conference on Neural Information Processing Systems (NeurIPS 2023).

of such a surrogate may stand-in for those of the black-box simulator, once the surrogate has been trained to approximate the simulator. Shirobokov et al. [34] take this approach in their method of *local generative surrogate optimisation* (L-GSO), which repeatedly trains local surrogate models to perform the inverse problem optimisation. For real-life applications, repeatedly training surrogates may be computationally challenging due to the need for complex surrogate models. Moreover, many scenarios demand optimising multiple *related* inverse problems, which strongly increases the computational burden for methods that start each optimisation from scratch. For instance, we may want to efficiently solve the muon detection inverse problem for many different potential input distributions over muon properties $\boldsymbol{x}$. In this work, we work towards efficiently solving inverse problems for settings where such environment distributions are not fixed, and where surrogate retraining is expensive. In particular:

1. We propose a novel surrogate-based inverse problem optimisation method that uses a single pre-trained surrogate, for settings where repeated local surrogate training is infeasible.
2. In particular, we train reinforcement learning (RL) policies to learn to switch between surrogate gradients and numerical simulator gradients during optimisation. Our experiments show that such policies can reduce simulator calls used during optimisation by up to 70%.
3. Additionally, using these policies, we can do successful optimisations on functions where using just numerical simulator gradients or just surrogate gradients fail, suggesting that the RL policies can learn to identify pitfalls in the optimisation and plan around them.

2 Background

Since black-box simulators are not amenable to automatic differentiation methods, Shirobokov et al. [34] propose to train local surrogate neural networks to mimic the simulator. This allows for optimisation over $\boldsymbol{\psi}$ with stochastic gradient descent, using only local surrogate gradients. Like them, we consider stochastic simulators of the form $\boldsymbol{y} = f_s(\boldsymbol{\psi}, \boldsymbol{x})$ where $\boldsymbol{y} \sim p(\boldsymbol{y}|\boldsymbol{\psi}, \boldsymbol{x})$ is a random variable and $\boldsymbol{x} \sim q(\boldsymbol{x})$ denotes a stochastic input. We are interested in minimising an expected observation loss function $\mathcal{L}$ with respect to the simulator parameters $\boldsymbol{\psi}$. Since exact evaluation of the expectation is not possible, we estimate it using N Monte Carlo samples:

$$\boldsymbol{\psi}^* = \arg\min_{\boldsymbol{\psi}} \mathbb{E}\left[\mathcal{L}(\boldsymbol{y})\right] = \arg\min_{\boldsymbol{\psi}} \int \mathcal{L}(\boldsymbol{y})\, p(\boldsymbol{y}|\boldsymbol{\psi}, \boldsymbol{x})\, q(\boldsymbol{x})\, d\boldsymbol{x}\, d\boldsymbol{y}, \tag{1}$$

$$\approx \arg\min_{\boldsymbol{\psi}} \frac{1}{N} \sum_{i=1}^{N} \mathcal{L}(f_s(\boldsymbol{\psi}, \boldsymbol{x}_i)). \tag{2}$$

After training a surrogate model $f_\phi : (\boldsymbol{\psi}, \boldsymbol{x}, \boldsymbol{z}) \to \boldsymbol{y}$ on data generated by f_s, we may use its gradients for stochastic gradient descent in $\boldsymbol{\psi}$. To account for the stochasticity of the simulator, the randomly sampled latent variable $\boldsymbol{z}$ is included in the surrogate input. Gradients are now computed as follows:

$$\nabla_{\boldsymbol{\psi}} \mathbb{E}\left[\mathcal{L}(\boldsymbol{y})\right] \approx \frac{1}{N} \sum_{i=1}^{N} \nabla_{\boldsymbol{\psi}}\, \mathcal{L}\left(f_\phi(\boldsymbol{\psi}, \boldsymbol{x}_i, \boldsymbol{z}_i)\right). \tag{3}$$

Running a simulator forward is often an expensive procedure, so practitioners aim to minimise the number of simulator calls required for the optimisation [34]. Surrogate gradient-based methods have been shown to outperform various baselines on this metric, such as optimisation based on numerical central-difference gradients [33], guided evolutionary strategies [22], Learning to Simulate [29], LAX gradients [13], and Cylindrical Kernel Bayesian Optimisation [25].

3 Method

Shirobokov et al. [34] iteratively optimise the inverse problem using the gradients of Equation 3. After each gradient step, they sample new data from the simulator and retrain the surrogate from scratch. They typically require 100 to 1000 of these retraining steps, which may be infeasible when the surrogate is a large neural network, which may be required for complex real-world applications. To that end, we propose to instead perform the optimisation using a single, pre-trained surrogate. While such a surrogate may be able to provide useful gradients in some part of the parameter space of $\boldsymbol{\psi}$, training it to have accurate gradients for the whole space is challenging. Thus, our goal should be

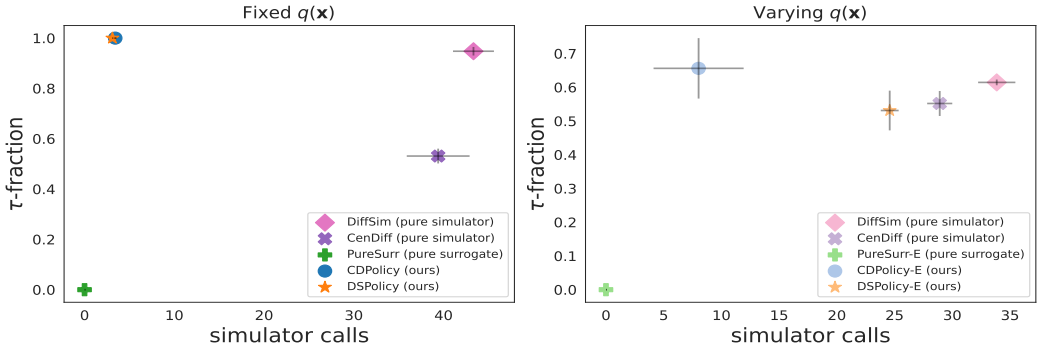


Figure 1: Pareto front for ThreeHump problem on simulator calls and τ-fraction. Upper-left corners of plots show the best-performing models. $q(\boldsymbol{x})$ is fixed for the left plot and changes every episode on the right, where a surrogate ensemble is used to capture surrogate uncertainty (denoted by "-E"). Evaluation is performed over 32 episodes per seed. Error bars denote the standard error of the mean (SEM) over 3 seeds. Note that PureSurr always fails to reach the target value.

to complement existing black-box optimisation approaches – such as numerical gradient estimation methods – with surrogate gradients, rather than to fully replace them. A natural next question is then how to switch between using surrogate gradients and numerical simulator gradient estimates. We propose to train a reinforcement learning policy to perform this switching task. Our method requires no surrogate retraining and a reduced number of simulator evaluations compared to L-GSO. In the following, we will describe such simulator evaluations as *simulator calls*.

We formalise the sequential optimisation as an episodic Markov Decision Process (MDP), and train our policy as an online actor-critic reinforcement learning agent with Proximal Policy Optimisation (PPO) [31] and Generalised Advantage Estimation (GAE) [30]. Our policy $\pi_\theta(a_t|s_t)$, parameterised by θ, at time t takes as input a state variable s_t and outputs actions a_t. The state is a tuple $(\boldsymbol{\psi}_t, t, l_t, \sigma_t)$, where $\boldsymbol{\psi}_t$ is the current parameter value, l_t is the number of simulator calls already performed this episode, and σ_t is some notion of uncertainty produced by the surrogate. Actions a_t consist of a binary random variable $b \in \{0, 1\}$, where 1 represents the decision to use gradients estimated from the simulator, i.e., do a simulator call. Action $b = 0$ then represents the decision to use the existing surrogate for gradient estimation. Transitions $T(s_t, a_t, s_{t+1})$ consist of a single gradient step using the Adam optimiser [19] with learning rate 0.1 using either simulator or surrogate gradients. Episodes end under three separate conditions. A: the optimisation reaches a parameter for which $\mathbb{E}\left[\mathcal{L}(\boldsymbol{y})\right]$ is below a target value τ (we call this *termination*), B: the maximum number of timesteps T has been reached, or C: the maximum number of simulator calls L has been reached. Since our goal is to reduce simulator calls, we give reward $r(s_t, a_t, s_{t+1}) = -1$ if $b = 1$ and reward 0 when $b = 0$. To incentivise termination, a reward penalty is added when B or C holds: the penalty is $-(L - l_t) - 1$ when B occurs and -1 when C occurs. This ensures the sum-of-rewards for non-terminating episodes is $-L-1$. We have observed that primarily using reward penalties based on l_t, rather than t, improves training stability.

During policy training, we solve the target inverse problem many times. Since applications often require solving multiple *related* inverse problems, we aim to exploit correlations in such families of inverse problems. In particular, we vary the input distribution $q(\boldsymbol{x})$ between episodes, which corresponds to, e.g., different potential input distributions over muon properties $\boldsymbol{x}$, such as incident angle and energy. Varying the inverse problem between episodes allows our policies to exploit such correlations and generalise to test-time problems.

Since the decision to use surrogate vs. simulator gradients should depend on surrogate quality, we condition the policy on surrogate uncertainty information σ. We construct σ as the standard deviation over predictions of an ensemble of surrogates. In particular, we train three surrogates on the same simulator data, using different seeds. Each of these surrogates performs $D = 100$ predictions $\boldsymbol{y}$ given input data $\{(\boldsymbol{\psi}, \boldsymbol{x}_i)\}_{i=1}^{D}$. We average these predictions per surrogate, as $\bar{\boldsymbol{y}} = \frac{1}{D}\sum_{i=1}^{D}\left[f_\phi(\boldsymbol{\psi}, \boldsymbol{x}_i)\right]$, and compute the standard deviation of these mean predictions over the ensemble. See Supplementary B for details.

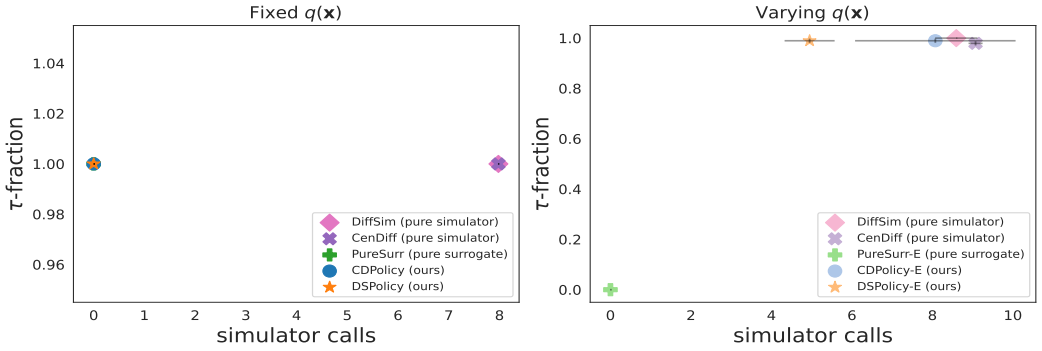


Figure 2: Pareto front for Rosenbrock problem on simulator calls and τ-fraction. Upper-left corners of plots show the best-performing models. $q(\boldsymbol{x})$ is fixed for the left plot and changes every episode on the right, where a surrogate ensemble is used to capture surrogate uncertainty (denoted by "-E"). Evaluation is performed over 32 episodes per seed. Error bars denote the standard error of the mean (SEM) over 3 seeds. For fixed $q(\boldsymbol{x})$, PureSurr, CDPolicy and DSPolicy all reach optimal performance of 0 simulator calls and τ-fraction 1. For varying $q(\boldsymbol{x})$, PureSurr always fails to reach the target value.

4 Experiments

We perform experiments on two stochastic simulator functions: the two-dimensional ThreeHump problem, and the ten-dimensional Rosenbrock problem (Supplementary B.1). Results are shown in Figures 1 and 2. Here we have plotted the Pareto front of base L-GSO, optimisation using only the surrogate (PureSurr), optimisation using only the simulator (DiffSim and CentrDiff), and our switching policies (DSPolicy and CDPolicy). DSPolicy uses the differentiable simulator for a simulator call, while CDPolicy uses central difference gradients instead. Our primary goal is to reduce the number of simulator calls for optimisation. Due to the stochastic nature of the optimisation, some optimisation episodes do not reach the target loss value within the allotted budget and have to be manually ended. We call the fraction of runs that reach the target loss value the 'τ-fraction' (plotted on the y-axis). On the x-axis, we plot the number of simulator calls; as such, points in the top-left correspond to better overall performance.

We use two different methods to obtain simulator gradients: numerical central-difference gradient estimation (for CentrDiff and CDPolicy and white-box access to the underlying function to create a *differentiable simulator* (for DiffSim and DSPolicy) using the reparameterisation trick [20] and straight-through gradient estimation [7]. Our policy methods generally outperform base L-GSO, pure simulator, and pure surrogate methods in both the ThreeHump and Rosenbrock problem, on both metrics. While there is no clear best-performer between central-difference and white-box gradient methods, adding the policy improves performance over pure-simulator methods in all cases. Our experiments show that using only gradients obtained from the simulator requires many simulator calls and can still lead to relatively low τ-fraction, for instance for the ThreeHump problem with changing $q(\boldsymbol{x})$. On the other hand, the pure surrogate method fails to terminate in almost all cases. This suggests that the policy is able to exploit the strengths of both the simulator and surrogate gradient estimates, avoiding the pitfalls of restricting the optimisation to either. We refer to Supplementary C for visualisations of resulting optimisation paths.

5 Conclusion and discussion

We have proposed a novel method for surrogate-based inverse problem optimisation of black-box simulators. Unlike previous work, our method does not require potentially expensive repeated surrogate retraining, while still aiming to minimise the number of *simulator calls*. Our reinforcement learning policies are able to do inverse problem optimisations with 10%-70% fewer simulator calls than L-GSO, using only a pre-trained surrogate model and a simple numerical gradient estimation method. Our results suggest that inverse problem optimisation may benefit from guidance with reinforcement learning agents. We refer to Supplementary A for a discussion of limitations.

References

[1] S. Agostinelli et al. GEANT4 – a simulation toolkit. *Nuclear Instruments and Methods in Physics Research Section A*, 2003.

[2] S. Alarie, C. Audet, A. E. Gheribi, M. Kokkolaras, and S. Le Digabel. Two decades of blackbox optimization applications. *EURO Journal on Computational Optimization*, 2021.

[3] T. Bakker, H. van Hoof, and M. Welling. Experimental design for MRI by greedy policy search. In *Advances in Neural Information Processing Systems*, 2020.

[4] T. Bakker, M. Muckley, A. Romero-Soriano, M. Drozdzal, and L. Pineda. On learning adaptive acquisition policies for undersampled multi-coil MRI reconstruction. In *Proceedings of Machine Learning Research*, 2022.

[5] W. Banzhaf, F. D. Francone, R. E. Keller, and P. Nordin. *Genetic Programming: An Introduction: On the Automatic Evolution of Computer Programs and Its Applications*. Morgan Kaufmann Publishers Inc., 1998.

[6] A. G. Baydin, K. Cranmer, P. de Castro Manzano, C. Delaere, D. Derkach, J. Donini, T. Dorigo, A. Giammanco, J. Kieseler, L. Layer, G. Louppe, F. Ratnikov, G. C. Strong, M. Tosi, A. Ustyuzhanin, P. Vischia, and H. Yarar. Toward machine learning optimization of experimental design. *Nuclear Physics News*, 2021.

[7] Y. Bengio, N. Léonard, and A. C. Courville. Estimating or propagating gradients through stochastic neurons for conditional computation. *arXiv preprint*, 2013.

[8] L. Bliek. A survey on sustainable surrogate-based optimisation. *Sustainability*, 2022.

[9] K. Cranmer, J. Brehmer, and G. Louppe. The frontier of simulation-based inference. *Proceedings of the National Academy of Sciences*, 2020.

[10] E. Daxberger, A. Makarova, M. Turchetta, and A. Krause. Mixed-variable bayesian optimization. In *Proceedings of the International Joint Conference on Artificial Intelligence*, 2020.

[11] F. de Avila Belbute-Peres, K. Smith, K. Allen, J. Tenenbaum, and J. Z. Kolter. End-to-end differentiable physics for learning and control. In *Advances in Neural Information Processing Systems*, 2018.

[12] J. Degrave, M. Hermans, J. Dambre, and F. wyffels. A differentiable physics engine for deep learning in robotics. *Frontiers in Neurorobotics*, 2019.

[13] W. Grathwohl, D. Choi, Y. Wu, G. Roeder, and D. Duvenaud. Backpropagation through the void: Optimizing control variates for black-box gradient estimation. In *Proceedings of the International Conference on Learning Representations*, 2018.

[14] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International Conference on Learning Representations, Workshop*, 2018.

[15] Y. Hu, J. Liu, A. Spielberg, J. B. Tenenbaum, W. T. Freeman, J. Wu, D. Rus, and W. Matusik. Chainqueen: A real-time differentiable physical simulator for soft robotics. In *International Conference on Robotics and Automation*, 2019.

[16] R. Iman, J. Davenport, and D. Zeigler. Latin hypercube sampling (program user's guide). [lhc, in fortran]. Technical report, Sandia Labs, 1980.

[17] E. Jang, S. Gu, and B. Poole. Categorical reparameterization with gumbel-softmax. In *Proceedings of the International Conference on Learning Representations*, 2017.

[18] E. Jonas. Deep imitation learning for molecular inverse problems. In *Advances in Neural Information Processing Systems*, 2019.

[19] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint*, 2014. doi: 10.48550/arXiv.1412.6980.

[20] D. P. Kingma and M. Welling. Auto-Encoding Variational Bayes. In *Proceedings of the International Conference on Learning Representations*, 2014.

[21] G. Lei, J. Zhu, Y. Guo, C. Liu, and B. Ma. A review of design optimization methods for electrical machines. *Energies*, 2017.

[22] N. Maheswaranathan, L. Metz, G. Tucker, D. Choi, and J. Sohl-Dickstein. Guided evolutionary strategies: augmenting random search with surrogate gradients. In *Proceedings of the International Conference on Machine Learning*, 2019.

[23] S. Mohamed, M. Rosca, M. Figurnov, and A. Mnih. Monte carlo gradient estimation in machine learning. *Journal of Machine Learning Research*, 2020.

[24] S. Neumann, S. Lim, A. G. Joseph, Y. Pan, A. White, and M. White. Greedy actor-critic: A new conditional cross-entropy method for policy improvement. In *Proceedings of the International Conference on Learning Representations*, 2023.

[25] C. Oh, E. Gavves, and M. Welling. BOCK : Bayesian optimization with cylindrical kernels. In *Proceedings of the International Conference on Machine Learning*, 2018.

[26] M. N. Omidvar, X. Li, and X. Yao. A review of population-based metaheuristics for large-scale black-box global optimization — part II. *IEEE Transactions on Evolutionary Computation*, 2022.

[27] T. Orekondy, P. Kumar, S. Kadambi, H. Ye, J. Soriaga, and A. Behboodi. WineRT: Towards neural ray tracing for wireless channel modelling and differentiable simulations. In *Proceedings of the International Conference on Learning Representations*, 2023.

[28] L. Pineda, S. Basu, A. Romero, R. Calandra, and M. Drozdzal. Active MR k-space sampling with reinforcement learning. In *Proceedings of the International Conference on Medical Image Computing and Computer-Assisted Intervention*, 2020.

[29] N. Ruiz, S. Schulter, and M. Chandraker. Learning to simulate. In *Proceedings of the International Conference on Learning Representations*, 2019.

[30] J. Schulman, P. Moritz, S. Levine, M. I. Jordan, and P. Abbeel. High-dimensional continuous control using generalized advantage estimation. In *Proceedings of the International Conference on Learning Representations*, 2016.

[31] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv preprint*, 2017. doi: 10.48550/arXiv.1707.06347.

[32] D. Schwalbe-Koda, A. R. Tan, and R. Gómez-Bombarelli. Differentiable sampling of molecular geometries with uncertainty-based adversarial attacks. In *Nature Communications*, 2021.

[33] H.-J. M. Shi, M. Q. Xuan, F. Oztoprak, and J. Nocedal. On the numerical performance of finite-difference-based methods for derivative-free optimization. *Optimization Methods and Software*, 2023.

[34] S. Shirobokov, V. Belavin, M. Kagan, A. Ustyuzhanin, and A. G. Baydin. Black-box optimization with local generative surrogates. In *Advances in Neural Information Processing Systems*, 2020.

[35] A. Stakia. Advanced multivariate analysis methods for use by the experiments at the large hadron collider. *Physica Scripta*, 2021.

[36] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 1992.

[37] T. Yin, Z. Wu, H. Sun, A. V. Dalca, Y. Yue, and K. L. Bouman. End-to-end sequential sampling and reconstruction for mri. In *Proceedings of Machine Learning for Health*, 2021.

A Limitations and future work

The primary limitation of our current work is the lack of evaluation on real-life simulators. We are currently aiming to apply our work to the muon detection particle physics problem mentioned in the main text, using the GEANT4 simulator [1].

In our experiments, we showed that optimisation using only differentiable simulators (obtained through white-box simulator access) can lead to poor performance on the ThreeHump problem, but not on the Rosenbrock problem. We suspect this is due to bias in straight-though estimation, which is used for ThreeHump, but absent in Rosenbrock. Using a different estimator for obtaining gradients of the binary sampling operation in ThreeHump, such as the Gumbel SoftMax estimator [17], may improve performance.

Gradient-based optimisation may get stuck in local optima of the loss surface $\mathbb{E}_{p(\boldsymbol{y}|\boldsymbol{\psi},\boldsymbol{x})}\left[\mathcal{L}(\boldsymbol{y})\right]$. We have already seen some evidence that the policy methods can learn to avoid pitfalls specific to the simulator and surrogate gradients by selecting when to use each: investigating what these pitfalls are and in which settings they occur is an interesting direction of future research.

Hyperparameter tuning has mostly involved reducing training variance through tuning the number of episodes used for a PPO iteration, as well as setting learning rates and the KL-threshold. Little effort has been spent optimising the policy or surrogate architectures; we expect doing so to further improve performance. Similarly, while PPO with a value function critic is a widely used algorithm, more recent algorithms may offer additional advantages, such as improved planning and off-policy learning for more data-efficient training [14; 24].

B Implementation details

B.1 Simulators

Our experiments use two stochastic simulator functions: ThreeHump and Rosenbrock. ThreeHump is a two-dimensional problem that lends itself well to visualisation, while the N-dimensional Rosenbrock problem is used to test our method in a higher-dimensional setting. We choose $N = 10$ in our implementation. Both of these simulators have scalar output values $\boldsymbol{y} = y$.

ThreeHump: Find the 2-dimensional $\boldsymbol{\psi}$ that optimises:[2]

$$\boldsymbol{\psi}^* = \arg\min_{\boldsymbol{\psi}} \mathbb{E}\left[\mathcal{L}(\boldsymbol{y}\right] = \arg\min_{\boldsymbol{\psi}} \mathbb{E}\left[\sigma(y-10) - \sigma(y)\right], \text{s.t.}$$
$$y \sim \mathcal{N}(y|\mu_i, 1),\ i \in \{1,2\},\ \mu_i \sim \mathcal{N}(x_i\, h(\boldsymbol{\psi}, 1),\ x_1 \sim U[-2,2],\ x_2 \sim U[0,5], \tag{4}$$
$$P(i=1) = \frac{\psi_1}{||\boldsymbol{\psi}||_2} = 1 - P(i=2),\ h(\boldsymbol{\psi} = 2\psi_1^2 - 1.05\psi_1^4 + \psi_1^6/6 + \psi_1\psi_2 + \psi_2^2.$$

We consider an episode terminated when $\mathbb{E}\left[\mathcal{L}(\boldsymbol{y}\right] = \frac{1}{N}\sum_{i=1}^{N}\mathcal{L}(f_s(\boldsymbol{\psi}, \boldsymbol{x}_i)) \leq \tau = -0.8$, which we evaluate after every optimisation step using $N = 10^4$ samples. Following [34], we use $\epsilon = 0.5$ as the trust-region size. The optimisation is initialised at $\boldsymbol{\psi}_0 = [2.0, 0.0]$; this is a symmetry point in the ThreeHump function such that optimisation with stochastic gradients can fall into either of the two wells around the two minima of the function. This requires our methods to learn good paths to both of these optima, making the task more interesting. In principle, optimisation could be initialised at any $\boldsymbol{\psi}_0$.

To test the generalisation behaviour of our method, we parameterise this target function further, by placing distributions on the bounds of the Uniform distributions that x_1 and x_2 are sampled from. We sample new bounds every episode, such that the policy sees multiple related – but different – simulators during training (and evaluation).

In particular, we sample lower and upper bounds of x_1 from respectively $\mathcal{N}(-2, 0.5)$ and $\mathcal{N}(2, 0.5)$. For x_2 we instead use $\mathcal{N}(0, 1)$ and $\mathcal{N}(5, 1)$. Note that this occasionally means episodes cannot terminate, as the specified termination value τ is below the minimum loss value for some samplings.

[2] Here the upper bound of x_1 and lower bound of x_2 are switched compared to the notation in Equation (3) of [34]. These bounds match the official implementation of L-GSO as of August 2023. Private correspondence with the authors has confirmed that this is the intended implementation.

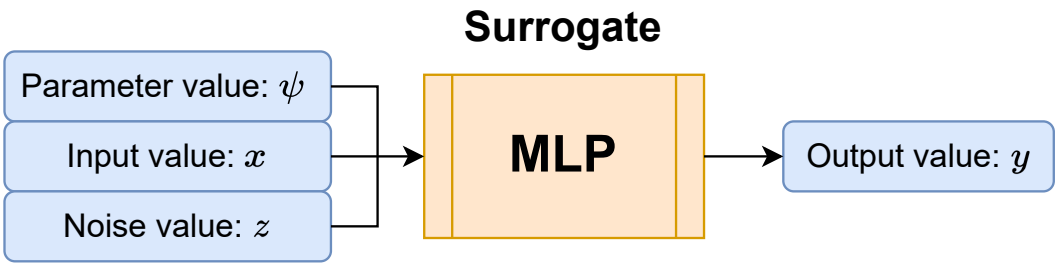


Figure 3: Schematic for the surrogate architecture. The surrogate is an MLP trained to mimic the simulator. It takes $(\boldsymbol{\psi}, \boldsymbol{x}, \boldsymbol{z})$ as input and outputs $\boldsymbol{y}$. The ensemble consists of 3 of these, trained on the same data with different seeds.

Rosenbrock: Find the N-dimensional $\boldsymbol{\psi}$ that optimises:

$$\boldsymbol{\psi}^* = \arg\min_{\boldsymbol{\psi}} \mathbb{E}\left[\mathcal{L}(\boldsymbol{y}\right] = \arg\min_{\boldsymbol{\psi}} \mathbb{E}\left[y\right], \text{s.t.}$$

$$y \sim \mathcal{N}\left(y \middle| \sum_{i=1}^{N-1} \left[(\psi_i - \psi_{i+1})^2 + (1-\psi_i)^2\right] + \mu, 1\right),\ \mu \sim N(\mu|x, 1),\ x \sim U[-10, 10]. \quad (5)$$

We consider an episode terminated when $\mathbb{E}\left[\mathcal{L}(\boldsymbol{y}\right] = \frac{1}{N}\sum_{i=1}^{N} \mathcal{L}(f_s(\boldsymbol{\psi}, \boldsymbol{x}_i)) \leq \tau = 3.0$, which we evaluate after every optimisation step using $N = 10^4$ samples. Following [34], we use $\epsilon = 0.2$ as the trust-region size. The optimisation is initialised at $\overrightarrow{2.0} \in \mathbb{R}^{10}$. When testing generalisation behaviour, we sample lower and upper bounds of x from respectively $\mathcal{N}(0, 2)$ and $\mathcal{N}(10, 2)$.

B.1.1 Simulator gradients

A simulator gradient step is performed in one of two ways, depending on the experiment:

DifferentiableSimulator: We make the simulators equation B.1 and equation B.1 differentiable using white-box knowledge of the simulator. Sampling operations from the Normal distributions are made differentiable using the reparameterisation trick [20]. Gradients for the binary sampling operation in equation B.1 are estimated with straight-through estimation [7], treating the sampling operation as a sigmoid with slope 10 as in [4; 37]. Gradients are computed by sampling 10^4 x-values from $q(\boldsymbol{x}$ for the current $\boldsymbol{\psi}$, doing a forward pass with the simulator, and differentiating w.r.t. $\boldsymbol{\psi}$ using automatic differentiation in PyTorch. A 'simulator call' thus consists of 10^4 function evaluations, as in [34]

CentralDifference: We use a simple central-difference gradient estimator to compute simulator gradients. We write $f_s'^{(\psi_p)}$ for the derivative of the simulator w.r.t. to the p-th element ψ_p of a k-dimensional $\boldsymbol{\psi}$. Then:

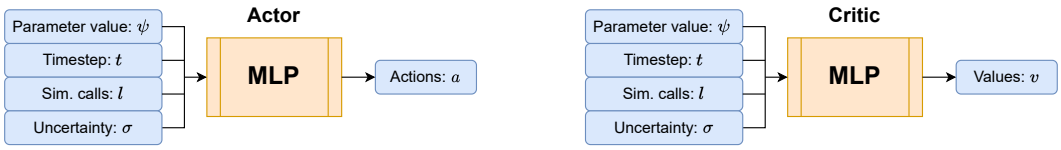
$$f_s'^{(\psi_p)} \approx \left(\bar{\mathcal{L}}(\psi_1, ..., \psi_p + \epsilon, ..., \psi_k) - \bar{\mathcal{L}}(\psi_1, ..., \psi_p - \epsilon, ..., \psi_k)\right)/2\epsilon, \quad (6)$$

where $\bar{\mathcal{L}} = \frac{1}{N}\sum_{i=1}^{N} \mathcal{L}(f_s(\boldsymbol{\psi}, x_i)),\ x_i \sim q(\boldsymbol{x})$. Following [34], we use $\epsilon = 0.1$ for estimating central-difference gradients. We set $N = 10^4$, such that a 'simulator call' consists of $2k \cdot 10^4$ function evaluations.

B.2 Surrogate

The surrogate consists of ReLU MLP with 2 hidden layers of 256 neurons that takes as input $(\boldsymbol{\psi}, \boldsymbol{x}, \boldsymbol{z})$ and outputs $\boldsymbol{y}$. $\boldsymbol{z}$ is sampled from a 100-dimensional diagonal unit Normal distribution. The surrogate architecture is schematically depicted in Figure 3.

Surrogates are trained on data generated from f_s. We sample M values of $\boldsymbol{\psi}_j$ inside a fixed-size box $U_\epsilon^{\boldsymbol{\psi}}$ – with sides 2ϵ – around the around the initial parameter $\boldsymbol{\psi}_0$ using an adapted Latin Hypercube sampling algorithm [16]. This follows the procedure in Shirobokov et al. [34] for training surrogates.



(a) Actor. (b) Critic.

Figure 4: Schematic for the policy architecture. The policy consists of a separate Actor and Critic, which are both MLPs. They take $(\boldsymbol{\psi}, t, l, \sigma)$ as input and output the actions and value function estimates. Actions are the decision b to do a simulator call or not.

We use and $M = 19$ for both the ThreeHump problem and the Rosenbrock problem. For the case of fixed $q(\boldsymbol{x})$, we sample a total $N = 10^4$ x-values for each of those $\boldsymbol{\psi}$ to pre-train the global surrogate. For varying $q(\boldsymbol{x})$, we instead sample 100 distributions, and sample 100 x-values from each of those distributions for each sampled $\boldsymbol{\psi}$, leading to $N = 10^4$ for this setting as well.

Surrogates are trained with the Adam optimiser for 10 epochs with a learning rate of 10^{-3} and batch size 512. The ensembles consist of 3 surrogates, each trained on the same data, but a different random seed. This pre-trained surrogate is then used throughout the entire optimisation procedure. A surrogate gradient step is performed by sampling 10^4 x-values from $q(\boldsymbol{x})$ for the current $\boldsymbol{\psi}$, doing a forward pass with the surrogate, and differentiating w.r.t. $\boldsymbol{\psi}$ using automatic differentiation in PyTorch.

B.3 Policy

The policy π_θ consists of a separate Actor and Critic neural network. Both are ReLU MLPs with a single hidden layer of 256 neurons, schematically depicted in Figure 4. Both networks take as input a tuple $(\boldsymbol{\psi}_t, t, l_t, \sigma_t)$, where $\boldsymbol{\psi}_t$ is the current parameter value (at timestep t), l_t is the number of simulator calls already performed this episode, and σ_t is the standard deviation over average surrogate predictions in the ensemble.

The Actor outputs a single value that is passed through a sigmoid activation and treated as a Bernoulli random variable from which b (the decision to do a simulator call or not) is sampled. The critic outputs a value-function estimate $V_\theta(s)$, where θ are policy parameters, that we use for computing advantage estimates in PPO. See section B.4 for details. Since rewards have unity order of magnitude, we expect return values to be anywhere in $[-T, 0]$. To prevent scaling issues, we multiply the critic output values by T before using them for advantage estimation.

B.4 Training

We train our policy in episodic fashion by accumulating sequential optimisation episodes and updating the policy using PPO [31] with GAE advantages [30] (discount factor $\gamma = 1.0$, GAE $\lambda = 0.95$. Episodes terminate once A: the target loss value τ has been reached, B: the number of timesteps $T = 1000$ has been reached, or C: the number of simulation calls $L = 50$ has been reached. Every training iteration we accumulate 4 episodes before doing PPO updates. We train for a total of 100 iterations.

Actor updates are performed using the PPO-clip objective (with clip value 0.2) on full trajectories with no entropy regularisation. We perform multiple Actor updates with the same experience until either the empirical KL-divergence between old and new policy reaches a threshold of $3 \cdot 10^{-3}$, or 20 updates have been performed. In practice, we rarely perform the full 20 updates. Updates use Adam with learning rate $3 \cdot 10^{-4}$.

Similarly, we perform multiple Critic updates using the Mean-Squared Error (MSE) between the estimated values $V_\theta(s_t)$ and the observed return (sum of rewards, as $\gamma = 1.0$) R_t at every timestep. We keep updating until either MSE ≤ 30.0 or 10 updates have been done. This helps the critic learn quickly initially and after seeing very surprising episodes, but prevents it from over-updating on very similar experience (as MSE will be low for those iterations). Updates use Adam with learning rate 10^{-4}.

See algorithm 1 for training pseudo-code. Evaluation is performed using algorithm 2 on 32 optimisation episodes.

Algorithm 1: Training the switching policy.

Data: Simulator $f_s(\boldsymbol{\psi}, \boldsymbol{x})$; pretrained surrogate $f_\phi(\boldsymbol{\psi}, \boldsymbol{x})$; observation loss function $\mathcal{L}$; policy π_θ; distributions $Q(q)$ over distributions $q(\boldsymbol{x})$ to sample $\boldsymbol{x}$ from; initial value $\boldsymbol{\psi}_0$; target function value τ; number T of timesteps to run each simulation for (episode-length); maximum number of simulator calls L; $\boldsymbol{\psi}$ optimiser OPTIM_ψ with learning rate λ; number of policy training iterations K; number of episodes to accumulate for a PPO step G; policy optimiser OPTIM_π; reward function $\mathcal{R}$; experience buffer B; discount factor γ.

for $k \in (1, ..., K)$ **do**
  Empty experience buffer B.
  for _ $\in (1, ..., G)$ **do**
    Initialise number of retraining steps done: $l \leftarrow 0$.
    Set return: $R \leftarrow 0$.
    Sample $\boldsymbol{x}$-distribution $q \sim Q$.
    for $t \in (1, ..., T)$ **do**
      Sample $\boldsymbol{x} \sim q(x)$.
      Obtain surrogate features σ (e.g., ensemble uncertainty) from surrogate $f_\phi(\boldsymbol{\psi}_t, \boldsymbol{x})$.
      Construct state: $s \leftarrow (\boldsymbol{\psi_t}, t, l, \sigma)$.
      Obtain action: $a = \texttt{use\_simulator} \leftarrow \pi_\theta(s)$.
      if *use_simulator* **then**
        Obtain simulator gradients $\boldsymbol{g}_t$ using numerical methods.
        Increment number of simulator calls: $l \leftarrow l + 1$.
      else
        Obtain surrogate gradients: $\boldsymbol{g}_t \leftarrow \nabla_\psi f_\phi(\boldsymbol{\psi}, \boldsymbol{x})|_{\boldsymbol{\psi}_t}$.
      Do optimisation step: $\boldsymbol{\psi}_{t+1} \leftarrow \text{OPTIM}_\psi(\boldsymbol{\psi}_t, \boldsymbol{g}_t, \lambda)$.
      `terminated` $\leftarrow \mathbb{E}\left[\mathcal{L}(f_s(\boldsymbol{\psi}_t, \boldsymbol{x})\right] \leq \tau$
      Obtain reward: $r \leftarrow \mathcal{R}(s, a, \boldsymbol{\psi}_{t+1})$.
      Store (s, a, r) and any other relevant information in buffer B.
      if *terminated* **then**
        `break`
      end
      if l *equals* L **then**
        `break`
      end
    end
  end
  Update policy $\pi_\theta \leftarrow \text{OPTIM}(\pi_\theta, B, \gamma)$.
end

C Additional analyses

Figures 5 and 6 depict examples of learned optimisation trajectories by the CDPolicy method on the ThreeHump problem. Example learning curves are depicted in Figure 7.

Algorithm 2: Inference with the switching policy.

Data: Simulator $f_s(\boldsymbol{\psi}, \boldsymbol{x})$; pretrained surrogate $f_\phi(\boldsymbol{\psi}, \boldsymbol{x})$; observation loss function $\mathcal{L}$; trained policy π_θ; distributions $q(\boldsymbol{x})$ to sample $\boldsymbol{x}$ from; initial value $\boldsymbol{\psi}_0$; target function value τ; number T of timesteps to run each simulation for (episode-length); maximum number of simulator calls L; $\boldsymbol{\psi}$ optimiser OPTIM_ψ with learning rate λ.

for $t \in (1, ..., T)$ **do**
- Initialise number of simulator calls done: $l \leftarrow 0$.
- Sample $\boldsymbol{x} \sim q(x)$.
- Obtain surrogate features σ (e.g., ensemble uncertainty) from surrogate $f_\phi(\boldsymbol{\psi}_t, \boldsymbol{x})$.
- Construct state: $s \leftarrow (\boldsymbol{\psi_t}, t, l, \sigma)$.
- Obtain action: $a = \texttt{use\_simulator} \leftarrow \pi_\theta(s)$.
- **if** *use_simulator* **then**
  - Obtain simulator gradients $\boldsymbol{g}_t$ using numerical methods.
  - Increment number of simulator calls: $l \leftarrow l + 1$.
- **else**
  - Obtain surrogate gradients: $\boldsymbol{g}_t \leftarrow \nabla_\psi f_\phi(\boldsymbol{\psi}, \boldsymbol{x})|_{\boldsymbol{\psi}_t}$.
- Do optimisation step: $\boldsymbol{\psi}_{t+1} \leftarrow \text{OPTIM}_\psi(\boldsymbol{\psi}_t, \boldsymbol{g}_t, \lambda)$.
- `terminated` $\leftarrow \mathbb{E}\left[\mathcal{L}(f_s(\boldsymbol{\psi}_t, \boldsymbol{x})\right] \leq \tau$
- **if** *terminated* **then**
  - `break`
- **end**
- **if** l *equals* L **then**
  - `break`
- **end**

end

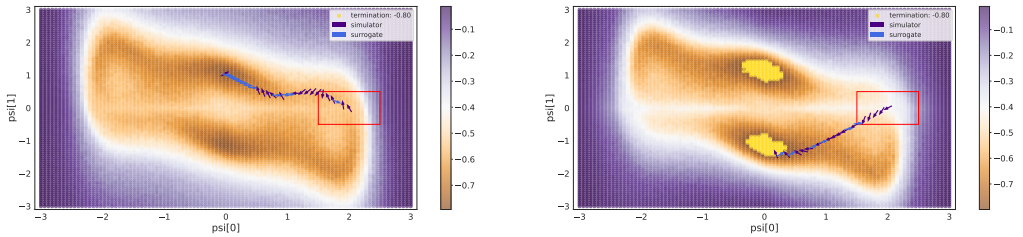


Figure 5: Examples of learned optimisation trajectories on a heatmap of the ThreeHump problem. Shown is $\frac{1}{N}\sum_{i=1}^{N}\mathcal{L}(f_s(\boldsymbol{\psi}, \boldsymbol{x}_i))$ for a grid of $\boldsymbol{\psi}$ values ($N = 100$). The yellow region denotes values of $\boldsymbol{\psi}$ that lead to termination. The $\epsilon = 0.5$ trust-region box around $\boldsymbol{\psi}_0$ is visualised by the red box. Trajectories are sampled from the final evaluation iteration of CDPolicy: shown here are two trajectories that terminate.

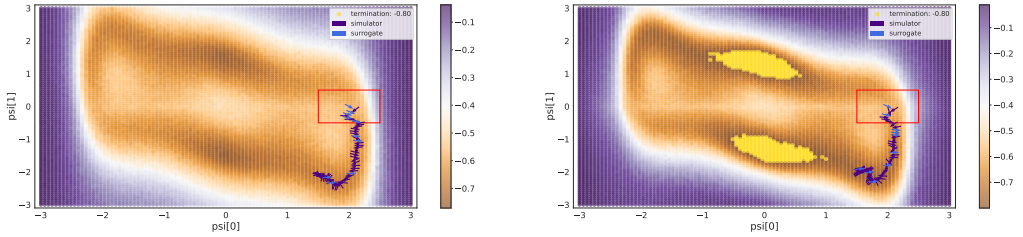
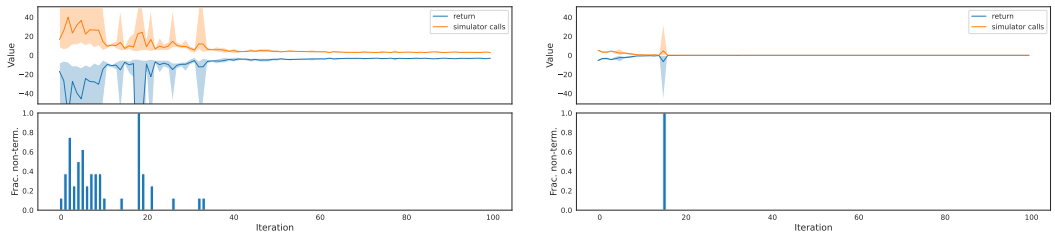


Figure 6: Examples of learned optimisation trajectories on a heatmap of the ThreeHump problem. Shown is $\frac{1}{N}\sum_{i=1}^{N}\mathcal{L}(f_s(\boldsymbol{\psi}, \boldsymbol{x}_i))$ for a grid of $\boldsymbol{\psi}$ values ($N = 100$). The yellow region denotes values of $\boldsymbol{\psi}$ that lead to termination. The $\epsilon = 0.5$ trust-region box around $\boldsymbol{\psi}_0$ is visualised by the red box. Trajectories are sampled from the final evaluation iteration of CDPolicy: shown here are two trajectories that do not terminate.



(a) ThreeHump.

(b) Rosenbrock.

Figure 7: Learning curves for CDPolicy on a) ThreeHump and b) Rosenbrock with fixed $q(\boldsymbol{x})$. The upper plots show average return and number of simulator calls per iteration (which consists of multiple episodes). Shaded regions are the min-max of all the episodes for an iteration. Bottom plots show the fraction of episodes that do *not* terminate in an iteration. Note that Rosenbrock seems much easier to learn than ThreeHump.