Selective Domain Randomization: A Parameter-Relevance Based Approach To Sim2Real

Francesco Capuano
Politecnico di Torino
s295366@studenti.polito.it

Francesco Pagano
Politecnico di Torino
s299266@studenti.polito.it

Francesca Mangone
Politecnico di Torino
s303489@studenti.polito.it

***Abstract*—In this work we use Reinforcement Learning to learn a policy able to solve a Continuous Control task in which the learning process completely takes place in a simulated environment. Further, we show how to effectively use Bayesian Optimization to overcome the reality gap and deploy the learned policy in the real world. Finally, we show a novel approach that aims at retrieving the relevance of the randomization of each parameter in the process used to obtain policy transferability.**

The source code for this project is available at `https://www.github.com/wibox/MLDLRL`.

***Index Terms*—Sim2Real, Deep Reinforcement Learning, Bayesian Optimization, Robotics, Hopper**

I. Introduction

This work is concerned with the control of a one-legged robot named "Hopper" [1]. The goal of the control task is to learn how to make the robot hop forward without falling while maximizing its horizontal speed.

The robot consists of four main body parts - the torso at the top, the thigh in the middle, the leg in the bottom, and a single foot on which the entire body rests. The four body parts are connected pairwise through actuable joints. The robot manages to move in the environment controlling the torque applied by such actuable joints. The focus of this work is related to solving the problem of identifying the optimal torque applied to each joint so to achieve the previously mentioned goal of hopping forward with maximal horizontal speed through Reinforcement Learning.

While this task might be tackled with the usage of traditional approaches mainly based on control theory, obtaining a reliable and correct description of the entirety of the dynamic previously described would obviously require a huge modelling effort. Nevertheless, even unmatched modelling efforts would deeply suffer the effects of the unavoidable under-modelling phenomenon.

In light of the promising results demonstrated in [2], in this work we investigate the application of Reinforcement Learning to solve the Hopper with in-simulation training.

II. Related Works

A. The Reinforcement Learning framework

Reinforcement Learning (RL) is the branch of Machine Learning devoted to investigate problems in which the focus is the prolonged-in-time interaction between programs and their environment.

Since RL-based approaches offer a way with which to tackle those problems whose focus is decision making in a hardly-to-model scenario, as observed in [3], this branch of Machine Learning is particularly interesting for the Robotics setting. RL-literature has produced a solid theoretical framework with which to tackle problems previously analysed with traditional control-theory techniques.

In the framework of RL, programs are represented by means of an *agent* which sees the environment in terms of its *state*. The interactions between the program and the environment are represented in terms of *actions* the agent performs on the environment, virtually as a consequence of having observed a particular state, and *rewards* the agent obtains as a consequence of both its condition and the state of the environment after an action has been performed.

RL-Robotics problems, such as the one we present in this work, are not solved by explicitly modelling the dynamics of the robot considered. They are solved considering a high-level description of the agent-environment interactions, that can be used in a trial-and-error learning process.

This high-level description of the agent-environment interactions is typically represented through a Markov Decision Process (MDP). Such a choice allows to:

- Embed uncertainty into the RL framework itself, since MDPs are stochastic processes.
- Solve even those problems for which only a description of the interaction is available.

B. MDPs

A model-free MDP $\mathcal{M}$ is described by means of a state space S, an action space A and a reward function $r : S \times A \times S \mapsto \mathbb{R}$.

The MDP with which the Hopper task is modeled is represented by:

- A state space S, in which each element s_t is a vector in $\mathbb{R}^{11}$ in which information related to the the position, height, overall and angular velocity at timestep t is presented.
- An action space A, in which each element a_t represents the torques applied by the actuators in the Hopper at time t. In this specific setting they are represented as a three-dimensional vector whose components are always in $[-1, +1]$.

- A reward function r. For this task, the reward function was shaped so as to incentivize the agent for not falling, not performing too large actions and hopping forward as fast as possible between two consecutive timesteps.

Once this setting has been laid out, the training process can be expressed as the process of obtaining a mapping from S to A that allows to select the most convenient action in each state considering some notion of long-term reward. The possibility of estimating such consequences depends on the aforementioned mapping from S to A, in RL referred to as *policy* $\pi : S \times A \mapsto [0, 1]$.

C. The Sim2Real framework

Once this problem's formulation is available, one could start the training process directly on real robotic hardware. However, this is not always a viable approach since it requires a huge amout on agent-evironment interactions which, on real hardware, are dangerous and expensive to obtain. As presented in [2], training the agent on a simulated environment solves this issue so as to obtain the possibility of controlling the amount of experience used.

However, while this solution does certainly solve relevant issues related to training in the real world, training in a simulated environment introduces a novel complex issue: how to transfer the policy learned in an approximate version of the reality to the real world?

The unavoidable difference between simulation and real world is often referred to as *reality gap*, whereas the field of study focused with those techniques with which learning policies able to overcome this reality gap is often referred as *sim2real*.

In this work we present sim2real techniques which are based on the usage of Domain Randomization as a way with which obtaining policies which are robust to the aforementioned reality gap.

III. Method

A. Policy Gradient Methods

In this work, we attempted at solving the task using four different algorithms: REINFORCE with Baseline, Actor-Critic, TRPO [4] and PPO [5].

1) REINFORCE: The REINFORCE algorithm is derived from the Policy Gradient Theorem, which formulates the gradient to be used in the process of training so as to maximize the expected return. However, this method is affected by slow learning and unstable results, because of the variance of the updates. A first improvement to this algorithm exploits a theoretical result allowing to introduce a baseline term in the gradient. This helps reducing and directing the variance of the policy updates. A typical choice for this constant is a standardized version of the return obtained after each timestep.

2) Actor-Critic: This algorithm deals with an issue which is overlooked in REINFORCE: there is no way of judging the goodness of the action that triggered the state transition since no notion of expected return is present in the baseline. Actor-Critic methods solve exactly this issue by introducing a baseline which estimates the current and next state value functions so as to critically chose the action to perform.

3) PPO & TRPO: These algorithms exploit an approximation of a surrogate objective function to perform conservative updates on the policy's parameters. TRPO does this by establishing a trust region in the neighborhood of the current policy's configuration. The trust region is obtained imposing a constraint on the KL-Divergence of two subsequent updates. On the other hand, PPO introduces a concept of proximity of the updates obtained clipping the objective function so as to avoid performing too large updates.

B. Domain Randomization (DR)

In this work, we simulated the reality gap considering two different simulated environments, a *source* and a *target* one. In both the environments the agent's goal was the same, but in the target environment the agent had to face the control of a different robot. In particular, the Hopper in the target environment have a torso mass shifted of 1 kg with respect to the source one.

When DR is applied, the agent is forced to learn to carry out the task not depending on the possibility of correctly estimating the parameters themselves, thus effectively learning a policy which is robust to changes, as the ones due to the reality gap, in these parameters. The measure of success in terms of policy transferability we adopted in this work is the average return over 50 different test episodes of the policy trained in simulation in the target environment, i.e. the source-target return.

In this work we used different techniques to perform domain randomization. Namely, we focused on:

1) Uniform Domain Randomization: Uniform Domain Randomization (UDR) is a DR technique that samples the values of the masses of the robot from a Uniform Distribution whose bounds are hand-tuned.

Once the values of the masses are sampled, the agent learns how to perform the task with respect to the current configuration to then reiterate the sampling process and, essentially, re-learn to perform the task with a different configuration. This forces the agent to learn to solve the task rather than to solve the environment.

However, such a simple approach implies a huge hyper-parameter tuning effort, since the bounds of the Distribution need to be fine tuned so as to observe success.

2) Adaptive Domain Randomization: A possible solution to the issue of having to fine tune the bounds of the probability distributions is to use Adaptive Domain Randomization (ADR) [6].

In this work, we particularly focused on the possibility of using Bayesian Optimization (BO) to perform ADR, as presented in [7].

Since the very objective of DR is to fine tune the parameters of the environment so as to obtain policy transferability (measured as source-target return), it is reasonable to find those parameters bound using information related to the actual performance of the policy on the target environment.

However, for the very same issues that were at the core of the necessity of introducing the sim2real framework, it is not always possible to verify the parameters direct impact on the target environment.

Bayesian Optimization for Domain Randomization (BayRN) is a solution which takes into account this crucial aspect being a very sample efficient technique. In particular, we focus on the application of BayRN to find the bounds $[\phi_{\text{min}}, \phi_{\text{max}}]$ for each of the source-parameters for which the source-target return is maximized.

3) Selective Domain Randomization: Being a gradient free algorithm, Bayesian Optimization might stall when applied in spaces whose dimensionality is not small. Moreover, it is not rare at all to consider parametrizations of the environment which are significantly larger than the one tackled in this work.

While the function $f : \Phi \mapsto \mathbb{R}$ mapping from the distribution parameters space to the source-target return is certainly a black-box function (whose gradients are not analytically available), one could use an approach based on finite-differences to approximate those gradient so as to implement solutions for ADR which are more robust than gradient-free approaches in optimizing f.

Alternatively, one could try to reduce the dimensionality of the parameters space in which randomization takes place, so as to be able to continue to exploit techniques such as BayRN. In this section, we present an algorithm which focuses exactly on this aspect.

Selective Domain Randomization (SDR) is an approach which aims at obtaining the relevance the randomization of each parameter has on the underlying dynamics. This method mainly leverages the assumption that, with all its limitations, the simulation's quality is high enough so that it is an approximation of the actual dynamics of the robot in the real world.

Assuming this, changes in the masses of the simulated robot, have an impact on the dynamics in the simulation in a way that can be linked to the effect they would have on the target environment dynamics.

The impact of the randomization of the simulation parameters is obtained considering a custom-defined metric we did elaborate to solve this task. In particular we focused on finding a way with which to capture the effect of the different parameters on the harshness of the training process.

To obtain such information, we exploited the assumption that the agent is able to learn a well performing action per each environment: if the agent capabilities are equal across all possible environments, then the changes in the training results are most likely due to the underlying dynamics the agent is trying to learn.

Therefore we designed a custom algorithm for the construction of a dataset in which to the different values of the actual masses of the robot was mapped a value of the metric m we defined to describe the training process. In particular, this metric was defined as $m = 0.4\bar{a} + 0.4\bar{r} + 0.2\bar{n}$, where:

- $\bar{r}$ is the average sum of rewards collected during the second half of the test episodes considered.

Fig. 1. Results obtained with REINFORCE

- $\bar{n}$ is the average number of timesteps in each episode during the second half of the episodes considered.
- $\bar{a}$ is the average (on the second half of training episodes) difference between the absolute values of the largest and smallest action. The underlying intuition is that the more the actions range is jerky, the less the agent is properly learning the task, since a_t is significantly and constantly different from a_{t+1}.

Once this data were collected, we obtained the relevance of each of the parameters we were performing randomization on analysing the feature importance (intended as mean decrease in impurity) of each parameter in the context of a Random Forest Regressor approximating the function mapping the mass values to m. Finally, we concluded to avoid randomizing the parameters having the smallest feature importance.

Furthermore, this approach may used to reduce the computational burden of using finite differentiation to perform ADR.

IV. Experiments

In this section we present the results we obtained during the experimentation phase.

A. REINFORCE

Both in light of [8] and our experiments, for what concerns the REINFORCE policy network we used a two layers neural network comprised of 64 hidden units each with Tanh activation to approximate the policy. The input of the network was the 11-dimensional representation of the state. In this case, the parametrization of the policy network is obtained considering the parameters of the network itself, which are updated backwarding according to the aforementioned Policy Gradient theorem. The forward pass of the Neural Network outputs a three-dimensional vector and a scalar used as mean and standard deviation respectively in a multivariate normal distribution from which the action are then sampled.

Figure 1 shows the return obtained in the considered timesteps for 5 different training routines using the default configuration presented in Table I. Considering that REINFORCE is known for being more an introductory algorithm than an effective solution for this task, we trained the agent with 20k

TABLE I
REINFORCE PARAMETERS

	Parameters tested	Default configuration	Best configuration
number of units	[32, 64]	64	32
gamma	[0.998, 0.999]	0.999	0.999
learning rate	[0.001, 0.005]	0.001	0.005
sigma	[0.25, 0.5, 1.25]	0.5	0.5
training episodes	50000	20000	50000

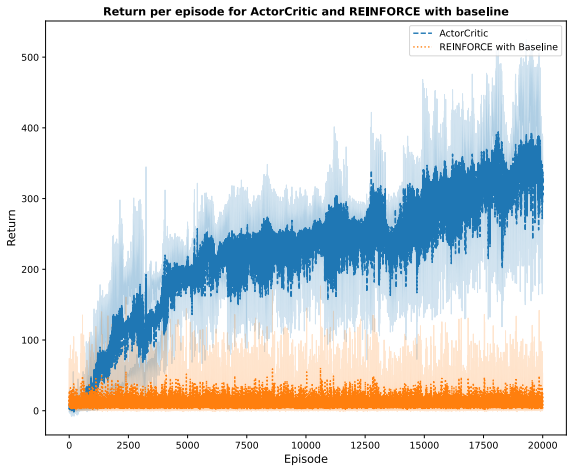

Fig. 2. Return per training episode for ActorCritic and REINFORCE with baseline

episodes only. However, even with this little experience it is still possible to observe how the mere presence of the baseline allows the agent to better direct the training process so as to obtain returns which are considerably higher than the ones obtained with the other two variants of REINFORCE that have been tested (namely, the vanilla and the reward-to-go one). The best configuration of parameters we found is presented in Table II.

B. Actor-Critic

The aforementioned estimate of the value function useful in the ActorCritic algorithm is obtained using a second Neural Network which rather than returning a probability distribution returns a scalar correspondent to the approximation $\hat{v}(s_t)$ of $v(s_t)$. This neural network is comprised of two layers consisting of 64 units each with tanh as activation function. For what concerns the loss function adopted, we used the MSELoss since the problem of approximating the state-value function could be tackled as a regression problem.

The approximation of $v(s_t)$ has been used as previously illustrated so at to guide the agent in its training process with policy updates which were more significant for actions having a significant effect on the return expected after the action with respect to the one anyway expected prior to performing such an action. Figure 2 shows the results we obtained in 5 randomized training routines. This figure clearly shows how the critic network allows the agent to drastically outperform REINFORCE in the same number of episodes.

TABLE II
ACTORCRITIC PARAMETERS

	parameters tested	Default configuration	Best configuration
batch size	[5, 10]	10	5
gamma	-	0.999	0.998
learning rate	[0.001, 0.005]	0.001	0.001
number of episodes	-	20000	20000
number of units	-	64	64
sigma	-	0.5	0.5

TABLE III
TRPO AND PPO PARAMETERS

	TRPO	
	Parameters tested	**Best configuration found**
policy	-	MlpPolicy
learning rate	[1e-3, 5e-3]	0.005
gamma	[0.998, 0.999]	0.999
target kl	[0.01, 0.05]	0.005
timesteps	-	150'000
activation function	-	tanh
batch size	[128, 256]	256
	PPO	
	Parameters tested	**Best configuration found**
policy	-	MlpPolicy
learning rate	[1e-3, 2.25e-3, 5e-3]	0.001
gamma	[0.998, 0.999]	0.998
target kl	[0.01, 0.05]	0.05
timesteps	-	150'000
activation function	-	tanh
batch size	[128, 256]	256

In this specific case we slightly modified the Actor Critic algorithm proposed in [9] for what concerns the frequency on the policy updates: ideally, the policy and critic network's parameters update should take place at each timestep of each episode.

However, this type of approach would result in updating so often the critic network that it never collect a consistent amount of experience. For this reason, we added a hyper-parameter to our implementation of the Actor-Critic algorithm related to the size of the batch of experience to collect. However, to be sufficiently close to the conceptual schema of the algorithm, the integer we ended up choosing as batch-size for the update has always been small throughout every step of the hyper-parameter tuning process.

Overall, it is possible to see how the agent learns the considered task up until a moment in which a bad update is taken. From that moment on, the agent starts re-building its experience for what is still possible. This type of degenerating behaviour is fully addressed by PPO and TRPO. Table II presents the hyper-parameter we experimented with and the best configuration found.

C. TRPO & PPO

For what concerns TRPO and PPO we resorted to the implementation presented in [10]. Both these algorithms were tested in their hyper-parameters as presented in Table III. This experiments were also carried out in light of the results presented in [8].

Figure 3 presents the results obtained experimenting with five different random seeds to vary the training routine.

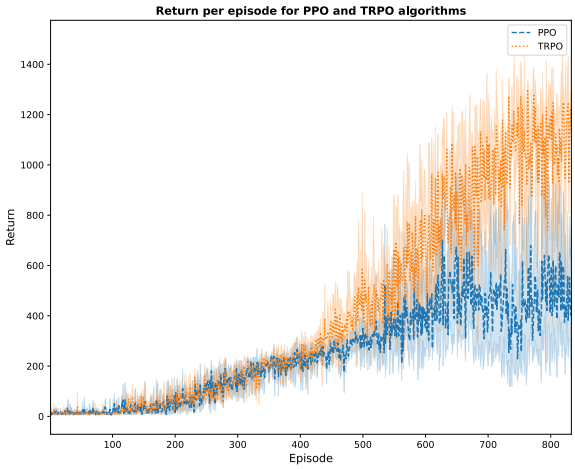

Fig. 3. Return per episode of training for PPO and TRPO

TABLE IV
COMPLETE COMPARISON AMONGST AGENTS

	Source-Source return	**Source-Target return**	**Target-Target return**
REINFORCE with Baseline	35.74	38.08	13.95
ActorCritic	353.6	362.6	343.5
PPO	569.2	541	934.1
TRPO	741.2	796.5	1001.3

It is clear not only how the degenerating behaviour previously observed is completely gone in this specific case, but also how TRPO outperforms each one of the four different approaches we used to solve the task.

Both the implementation of TRPO and PPO use as quantity to control the training experience the total, inter-episode, number of timesteps the agent is allowed to use for training. To better compare these two algorithms with REINFORCE and Actor-Critic, here we present the different learning curves as a function of the episodes of training rather than the allowed timesteps.

Our experiments did once more prove how significant the relative dissimilarity really is in this case since, in less that one thousand episodes, the agent manages to obtain results which are three times as good as the ones obtained with Actor-Critic.

The four agents have been evaluated with respect to the average return over 50 test episodes when both the training and testing took place in the source environment, when the agent was trained in the source environment and tested in the target one and, lastly, when both training and testing took place in the target environment. A complete comparison of our results is presented in Table IV.

D. Uniform Domain Randomization

Once the agents have been shown to solve the hopper task in the simulated environment, we focused on obtaining Domain Transferability using Domain Randomization. In particular, in this section we present our experiments both varying parameters of the environment and parameters of the environment.

This two-levels experimentation was done in the sake of obtaining the best possible set of parameters for the task at hand in a sim2real perspective. We experimented with various levels of the masses and with different hyperparmeters. Since TRPO outperformed the other algorithms, we mainly focused on carrying out experiments involving it. Table V shows the parameters we experimented with and the solution we obtained.

Using our best configuration, we observed a Source-Source average return of 1181.6 and a Source-Target average return of 1092.5.

E. Adaptive Domain Randomization

The bounds of the probability distributions used as a way of obtain a robustness of the policy to the reality gap can also be obtained as output of the process of Bayesian Optimization. As per what presented in [7], we resorted to the Botorch [11] library to perform such a task. The parameters we used for our experiments are presented in Table VI.

With these parameters we obtain as bounds for the probability distribution of the parameters presented in Table VII.

With these bounds, we observed a Source-Source average return of 1218.65 ± 272.54 and a Source-Target average return of 1440.50 ± 143.65.

F. Selective Domain Randomization

To obtain the relevance of each parameter, we first built a set of 100 observations which were mapping to each value of the masses considered the corresponding value of the metric m.

Once this empirical evidence has been obtained, we trained a Random Forest Regressor (in the scikit-learn implementation) and then retrieved the feature importance of each column using. The results we obtained are presented in Table VIII.

TABLE V
UDR BOUNDS

	Parameters tested	**Best configuration**
thigh lower bound (kg)	[2, 4]	2
thigh upper bound (kg)	[4.5, 5.5]	4.5
leg lower bound (kg)	[3, 7]	3
leg upper bound (kg)	[4.5, 10]	4.5
foot lower bound (kg)	[1.5, 5]	1.5
foot upper bound (kg)	[6, 9]	9

TABLE VI
BAYRN PARAMETERS

	Parameters tested
number of rollouts	5
general lower bound (kg)	0.25
general upper bound (kg)	10
number of init iterations	5
maximal number of iterations	15
timesteps	250'000

TABLE VII
BAYRN DISTRIBUTION PARAMETERS BOUND

	Value
thigh lower bound (kg)	5.30
thigh upper bound (kg)	5.56
leg lower bound (kg)	1.00
leg upper bound (kg)	4.55
foot lower bound (kg)	0.97
foot upper bound (kg)	8.75

TABLE VIII
FEATURE IMPORTANCE OF EACH PARAMETER

	feature importance
thigh mass	0.5779
leg mass	0.1705
foot mass	0.2516

As it is possible to see, the *leg mass* had the smallest feature importance value, even though its magnitude is still very much relevant being approximately 30% less relevant than the second least important feature. However, to validate our reasoning we applied BayRN considering to randomizing every part's mass but the *leg*.

Table IX shows the average 50 episodes Source-Target return obtained when randomizing all the masses versus when randomizing only those related to the largest feature importance.

TABLE IX
ADR VS. SDR RESULTS

Randomized masses	**Source-Target return**
[thigh, leg, foot]	1440.50
[thigh, foot]	540.16

As it is possible to see, this reasoning shows how relevant the masses are, since the value of the Source-Target return obtained neglecting the randomization of *leg* is approximately one third of the one obtained randomizing all the masses.

V. CONCLUSIONS AND FUTURE WORKS

In this work we did apply the Reinforcement Learning framework to solve the optimal control task of the Hopper robot. We implemented from scratch REINFORCE and Actor-Critic and observed how neglecting the concept of conservative updates highly affects these algorithms' performance.

When compared with algorithms concerned with the key concept of relative dissimilarity, these algorithms were outperformed, particularly by TRPO, which we mainly used in our experiments concerned with Domain Randomization for policy transferability.

We first hand-tuned the probability bounds and found the set of bounds yielding the largest Source-Target average return. Then, we moved to a more analytical approach using Bayesian Optimization to find the best set of bounds in a predefined range using an approximation of the Source-Target average return as metric of success. However, considering that such a method would risk to stall when the dimensionality of the search space increases, we implemented a novel framework with which to retrieve the relevance of the feature to randomize using a classic Random Forest Regressor, so as to reconstruct the link between each mass and a metric characteristic of the harshness of the training process.

Such an approach could be used in all those contexts in which the relevance of the parameters to randomize is to be taken into account, also considering how randomizing unnecessarily some parameters could be way worse than sub-optimal.

As future work, we plan on further investigating the employment of this solution in those use cases in which finite differentiation can be used to perform ADR. Moreover, we plan on investigating how this approach could be used in those simulated environments which are highly aligned to the real one, such as the ones obtained with techniques shown in DROPO. In this case, we believe the results of SDR would be more significant, because of the increase in the quality of the simulation (upon which SDR relies).

In principle, resorting to SDR for finite differentiation, one could drastically increase the quality of the found solution, considering the much more robust theory behind gradient-based optimization as opposed to gradient-free one.

REFERENCES

[1] E. Todorov, T. Erez, and Y. Tassa, "Mujoco: A physics engine for model-based control," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 5026–5033, 2012.

[2] R. Antonova, S. Cruciani, C. Smith, and D. Kragic, "Reinforcement learning for pivoting task," *arXiv preprint arXiv:1703.00472*, 2017.

[3] J. Kober, J. A. Bagnell, and J. Peters, "Reinforcement learning in robotics: A survey," *The International Journal of Robotics Research*, vol. 32, no. 11, pp. 1238–1274, 2013.

[4] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in *International conference on machine learning*, pp. 1889–1897, PMLR, 2015.

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

[6] I. Akkaya, M. Andrychowicz, M. Chociej, M. Litwin, B. McGrew, A. Petron, A. Paino, M. Plappert, G. Powell, R. Ribas, *et al.*, "Solving rubik's cube with a robot hand," *arXiv preprint arXiv:1910.07113*, 2019.

[7] F. Muratore, C. Eilers, M. Gienger, and J. Peters, "Data-efficient domain randomization with bayesian optimization," *IEEE Robotics and Automation Letters*, vol. 6, no. 2, pp. 911–918, 2021.

[8] P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger, "Deep reinforcement learning that matters," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 32, 2018.

[9] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[10] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, and N. Dormann, "Stable-baselines3: Reliable reinforcement learning implementations," *Journal of Machine Learning Research*, vol. 22, no. 268, pp. 1–8, 2021.

[11] M. Balandat, B. Karrer, D. R. Jiang, S. Daulton, B. Letham, A. G. Wilson, and E. Bakshy, "BoTorch: A Framework for Efficient Monte-Carlo Bayesian Optimization," in *Advances in Neural Information Processing Systems 33*, 2020.