Generalizing Learned Policies to Unseen Environments using Meta Strategy Optimization

Jeremiah Coholich

Abstract

We attempt to reproduce the claims in [9]. The authors train a quadruped robot in simulation to walk forward with a Deep Reinforcement Learning algorithm called Meta Strategy Optimization (MSO). By conditioning the learned policy on a small latent space, the authors are able to adapt the learned policy to environments that differ significantly from the training environment, beating other generalization methods including Domain Randomization. Using MSO, we are able to achieve a higher reward than DR on the training environment, but are not able to generalize to the test environments as well as DR.

1. Introduction

Deep reinforcement learning (DRL) algorithms have been used to learn behaviors for robots in recent years [5][7][2][10]. However, these methods are typically sample-inefficient and require lots of data. Because of this, policies are usually trained in simulation, since collecting data on a physical robot is time-consuming and potentially dangerous given the stochastic nature of many DRL algorithms. However, a policy learned in simulation may not transfer well to the real robot due to shifts in dynamics, sensor noise, and other or unknown modeling inaccuracies. DRL researchers have come up with several methods of enabling DRL models trained in simulation to generalize to other similar environments, including Domain Randomization (DR) [5], Strategy Optimization [8], and Meta Strategy Optimization (MSO), to name a few [9].

In this study, we replicate some of the claims in the paper *Learning Fast with Meta Strategy Optimization* by Yu, Tan, Fai, Coumans, and Ha [9]. In this paper, the authors train a DRL algorithm called Meta Strategy Optimization on a simulated Ghost Minitaur quadruped (Fig. 1) with the goal of teaching it to walk forwards. They then use the same algorithm to adapt the learned policy to numerous test environments, both simulated and real, which have significant shifts in physics and robot parameters not seen in training. They compare the performance of MSO to Domain Randomization and Strategy Optimization with a Projected Universal

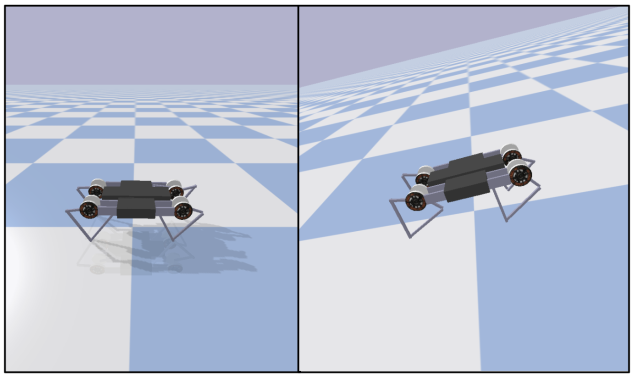
Figure 1. The Ghost Minitaur training environment (left) and one of the test environments (right), where the robot must walk up a slope, which is never encountered during training. Both environments randomize physics and robot parameters.

Policy (SO-PUP). MSO has the been generalization performance.

The key concept behind MSO is conditioning the policy on a learned latent space, which is optimized for every set of randomized simulation parameters. Unlike the parameterized policy, the latent space is low dimensional (two-dimensional in our experiments) and can be learned very quickly, in a few iterations. The latent space is not directly conditioned on the parameters of the simulation, but found by freezing the policy parameters and directly optimizing the latent space to achieve the highest reward in the training environment. After that, the latent space is frozen and concatenated with the observation at every step of during data collection for policy training.

In contrast, domain randomization learns a regular, flat policy without any latent variables. The same policy is used with any set of simulation parameters. The policy must learn to be robust or invariant to the simulation parameters that are randomized. Typically, polices learned with domain randomization are conservative and robust.

To limit the scope of this replication study, we test the trained policy on only two test environments in simulation only. We also only compare Meta Strategy Optimization to Domain Randomization. [9] shows that the DRL algorithms

Parameter	**Training**	**Extended Randomization**
Mass	60% - 160%	45% - 175%
Battery Voltage	10V - 18V	8.8V - 19.2V
Motor Viscous Damping	0 - 0.02 (N-m-s)	0 - 0.026 (N-m-s)
Contact Friction	0.2 - 1.25	0.1 - 1.365
Motor Strength	50% - 150%	35% - 165%
Latency	0 - 80 ms	0 - 100 ms
Robot Link Inertias	25% - 200%	25% - 1.525%

Table 1. Parameter randomization ranges for the training environment and the extended randomization test environment

that are able to adapt to different simulated environments also adapt the best to the physical robot. Therefore, ensuring that a policy can generalize well in simulation is a good first step before testing it on a real robot.

2. Approach

PyBullet, a free and open-source physics simulator, was used to create the quadruped reinforcement learning environment [1]. All training and test environments were created by modifying the third-party OpenAI Gym Environment ”MinitaurBulletEnv-v0”, whose source code is included in the PyBullet repository. The training environment was designed to match the training environment specified in [9] as closely as possible. However, we run our training simulation at 100 Hz instead of 50 Hz, since the PyBullet documentation warns that various simulation parameters will have to be re-tuned if the time step changes from the default of 0.01 seconds. The first of two test environments employs extended randomization of simulation parameters, with a range increase of 30% for all parameters. The randomized parameter ranges are given in table 1. The second test environment employs the same randomization ranges as the training environment, but the robot is tasked with walking up a slope instead of a flat plane (see Fig. 1). The angle of the slope is sampled uniformly from [5, 15] degrees.

The reward function used the train the quadruped in [9] is given in Eq. 1

$$r = \text{clip}\left((\mathbf{p}_n - \mathbf{p}_{n-1}) \cdot \mathbf{d}/dt,\ -1.0m/s,\ 1.0m/s\right) \quad (1)$$

However, upon review of the reported returns in [9] and the videos provided by the authors, it seems clear that Eq. 1 has a mistake, and the reward function in Eq. 2 was actually used, which is independent of the simulation timestep.

$$r = \text{clip}\left((\mathbf{p}_n - \mathbf{p}_{n-1}) \cdot \mathbf{d},\ -0.02m,\ 0.02m\right) \quad (2)$$

We employ the same reward function as Eq. 2, minus the clipping, since we are not concerned about the safety of a physical robot. The returns reported in [9] should be directly comparable to our returns since we also limit our episodes to 5 seconds. It is worth noting that summing the returns of Eq. 2 across all timesteps of an episodes gives the distance traveled by the robot in meters.

The authors of [9] use derivative-free optimization methods to find the best latent variables for every environment instance and to optimize the policy network. However, these methods are generally known to be slower and less sample-efficient than derivative-based optimization methods. Because of this, we used Proximal Policy Optimization (PPO) with Adam [3] to optimize the neural network that parameterizes the policy and value functions (a combined ”actor-critic” network). PPO is a state-of-the-art DRL algorithm which has shown success in many domains [6]. A publicly-available PyTorch implementation of PPO was used and modified for our experiments [4]. We used the default PPO hyperparameters of the implementation because they are able achieve a high reward on the unmodified ”MinitaruBulletEnv-v0” environment. For ease of implementation, we retained the use of derivative-free methods for searching the latent space, which is only two-dimensional. However, we further simplify by searching via random sampling instead of Bayesian Optimization. Just like in the MSO paper, we limit the number of episodes to 25 when searching the latent space.

In [9], the MSO training algorithm samples from 5 randomized environment instances per policy update, each with a unique set of simulation parameters and corresponding learned latent space. However, due to difficulties in modifying the PPO implementation to support unique parallel environments, our training algorithm is limited to only one randomized environment instance per update. The simulation parameters are randomized after every update, after which a new latent space is learned. In our domain randomization experiments, 10 environments are trained in parallel to speed up training. Each environment has its own set of simulation parameters that are randomized again when an episode finishes. We train both MSO and DR for 1e6 samples on eight different random seeds each.

The value of our latent variables during training is confined to [-1.0, 1.0]. During testing, we try MSO with a latent variable range of [-2.0, 2.0], because we got poor results

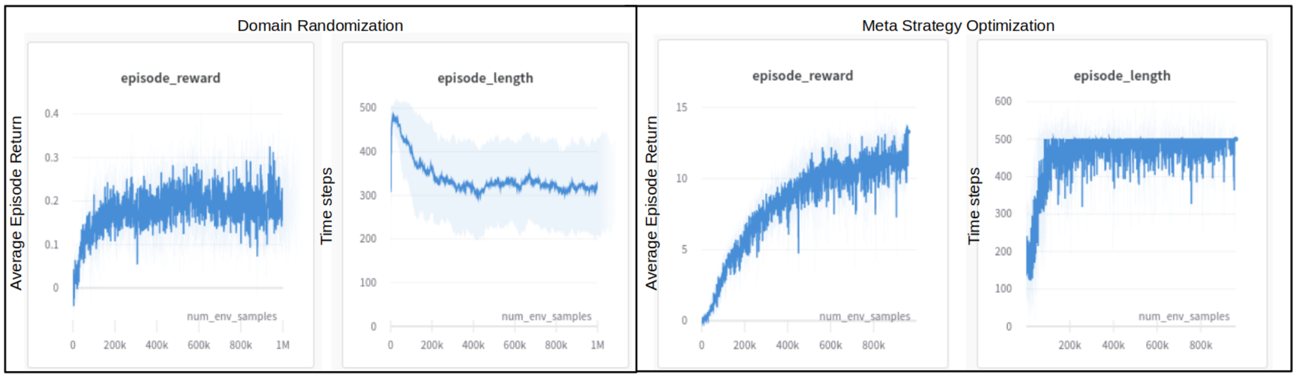

Figure 2. Training curves for Domain Randomization and Meta Strategy Optimization. The latter achieves a much higher reward in the training environment. Each curves is the average of eight random seeds. The shaded background curve represents the standard deviation.

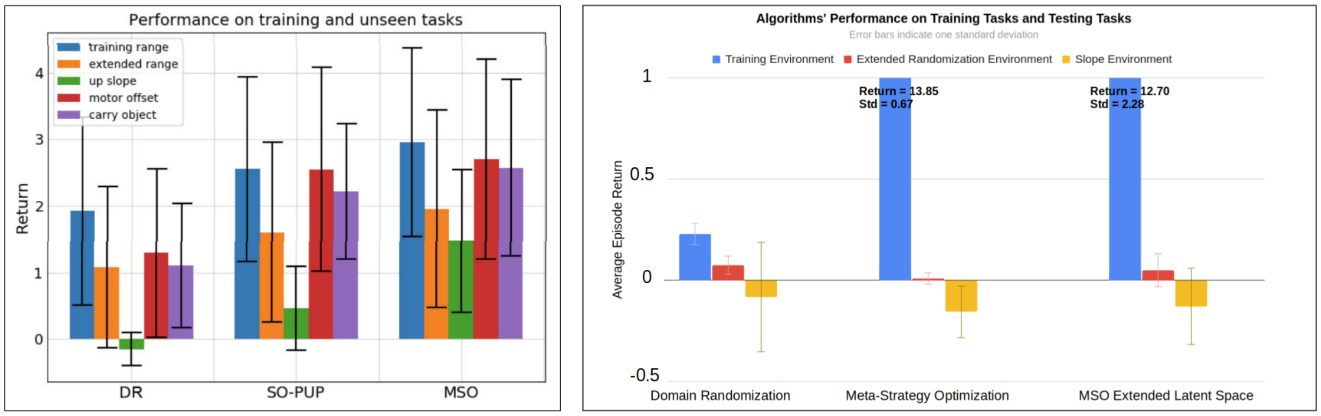

Figure 3. A comparison of the results shown in [9] (left) to the results from our study (right). The error bars represent one standard deviation. Our returns are calculated using the average of 10 random seeds. Note that vertical axis of the right plot stops at 1.0, although some returns are much higher.

with the MSO-learned policy on the test environments, as seen in the following section. We hypothesized that perhaps the latent variable range needs to be expanded in order for the agent to perform well on environments with parameters outside of the training ranges. The authors of [9] do not disclose information about any bounds of the latent space.

3. Results and Discussion

As seen in Fig. 2, the Domain Randomization reward plateaus at about 0.2 at 200k samples, failing to improve for the rest of the 800k samples. However, the MSO policy continues to improve even at 1e6 samples, achieving a reward of about 13.75. As seen in Fig. 4, the DR policy only learns to take one step forward in the training environment before falling over and thus triggering the simulation to terminate. The MSO policy learns to quickly run forwards.

We achieve a much higher reward for meta strategy optimization than the authors of [9] do. This is only partially explained by the lack of clipping on the reward function. The maximum cumulative reward using Eq. 2 for a 5 second episode is 5.0. As shown in Fig. 3, [9] fails to even get within one standard deviation of the maximum reward. However, our MSO algorithm achieves a reward of 13.85 on the training environment, which is about 360% higher. Presumably, using PPO with Adam resulted in better sample efficiency and faster learning.

However, perhaps our MSO algorithm implementation achieved superior performance on the training environment through over-fitting – it performs much worse on the two test environments and does not beat domain randomization. MSO with the extended latent space range does better than MSO, but still not better than Domain Randomization. Additionally, perhaps random search is not as effective in finding optimal latent variables as is Bayesian Optimization.

4. Conclusion

We were not able to replicate the Meta Strategy Optimization algorithm's ability to generalize. However, we did

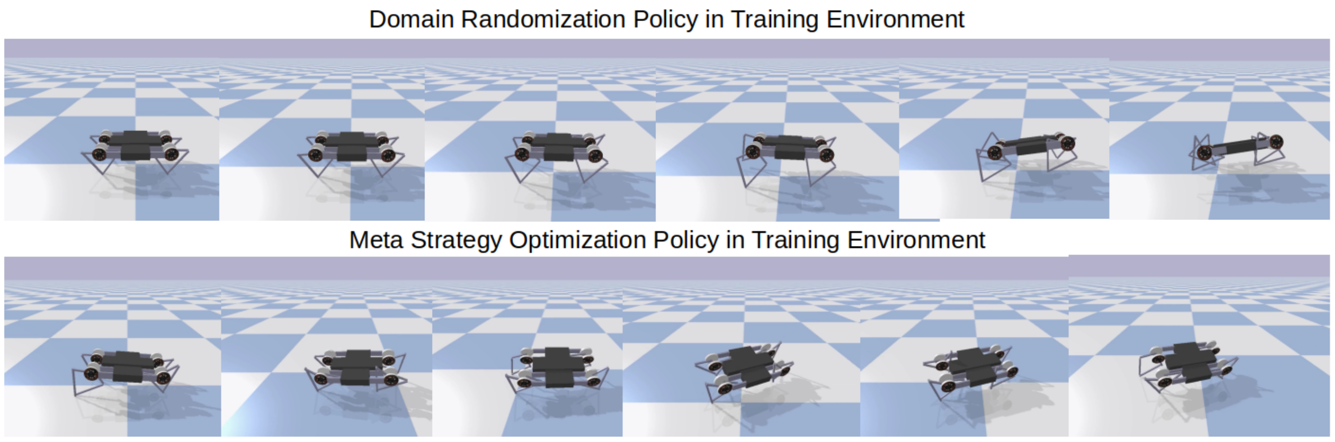
Domain Randomization Policy in Training Environment
Meta Strategy Optimization Policy in Training Environment

Figure 4. The two learned policies in the training environment. The DR policy takes one step and falls over. The MSO policy runs very fast comparatively; the frequency of individual steps is faster than the frame rate. Each tile is one meter squared.

find that it was able to achieve a much higher reward than Domain Randomization on the randomized training environment. However, this is not quite a fair comparison due to the fact the the MSO policy is allowed to optimize over prior information (training the latent variables over 25 episodes) before being evaluated. However, reinforcement learning is generally very sensitive to hyperparameters and perhaps a different setting of learning rate, PPO epochs, etc. would yield results more consistent with [9]. Given our findings so far, we would not feel it is worth it to implement the MSO on a real quadruped robot yet.

References

[1] Erwin Coumans and Yunfei Bai. Pybullet, a python module for physics simulation for games, robotics and machine learning. `http://pybullet.org`, 2016–2019.

[2] Shixiang Gu, Ethan Holly, Timothy Lillicrap, and Sergey Levine. Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. In *2017 IEEE international conference on robotics and automation (ICRA)*, pages 3389–3396. IEEE, 2017.

[3] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[4] Ilya Kostrikov. Pytorch implementations of reinforcement learning algorithms. `https://github.com/ikostrikov/pytorch-a2c-ppo-acktr-gail`, 2018.

[5] Xue Bin Peng, Marcin Andrychowicz, Wojciech Zaremba, and Pieter Abbeel. Sim-to-real transfer of robotic control with dynamics randomization. In *2018 IEEE international conference on robotics and automation (ICRA)*, pages 1–8. IEEE, 2018.

[6] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[7] Jie Tan, Tingnan Zhang, Erwin Coumans, Atil Iscen, Yunfei Bai, Danijar Hafner, Steven Bohez, and Vincent Vanhoucke. Sim-to-real: Learning agile locomotion for quadruped robots. *arXiv preprint arXiv:1804.10332*, 2018.

[8] Wenhao Yu, C Karen Liu, and Greg Turk. Policy transfer with strategy optimization. *arXiv preprint arXiv:1810.05751*, 2018.

[9] Wenhao Yu, Jie Tan, Yunfei Bai, Erwin Coumans, and Sehoon Ha. Learning fast adaptation with meta strategy optimization. *IEEE Robotics and Automation Letters*, 5(2):2950–2957, 2020.

[10] Fangyi Zhang, Jürgen Leitner, Michael Milford, Ben Upcroft, and Peter Corke. Towards vision-based deep reinforcement learning for robotic motion control. *arXiv preprint arXiv:1511.03791*, 2015.

5. Appendix

Hyperparameters used in the PPO algorithm when training MSO policies:
Number of Parallel Environments: 1
Samples per environment per update: 2048
Learning Rate: 3e-4
Entropy Coefficient: 0.0
Value Loss Coefficient: 0.5
PPO Epochs: 10
MiniBatches: 32
Discount Factor: 0.99
Lambda Parameter for Generalized-Advantage-Estimation: 0.95
Total Training Steps: 1e6
Use Linear Learning Rate Decay: True
Use CUDA: False
Seed: (varied between runs)

Hyperparameters used in the PPO algorithm when training DR policies:
Number of Parallel Environments: 10
Samples per environment per update: 205
Learning Rate: 3e-4
Entropy Coefficient: 0.0
Value Loss Coefficient: 0.5
PPO Epochs: 10
MiniBatches: 32
Discount Factor: 0.99
Lambda Parameter for Generalized-Advantage-Estimation: 0.95
Total Training Steps: 1e6
Use Linear Learning Rate Decay: True
Use CUDA: False
Seed: (varied between runs)

Run time for training an MSO policy is about 2.5 hours for DR policies and 7 hours for MSO policies with up to 8 instances running simultaneously on a Ryzen 3700x CPU.

Eight runs were used for training, while 10 runs were used obtaining the results displayed in Fig. 3.

Pybullet can be installed at the hyperlink in [1].