DEEP REINFORCEMENT LEARNING POLICY GRADIENT METHODS FOR QUADRUPEDAL ROBOT GAIT TRAJECTORY PLANNING AND CONTROL

Alexander Krolicki*, **Sarang Sutavani**†
Clemson University
Clemson, SC 29634 USA

ABSTRACT

The focus of this paper is to provide insight regarding the importance of policy gradient methods when applied to continuous systems. We study the development of policy gradient algorithms, and explain when and how policy gradients should be applied or when they are inadequate for the given problem. We specifically investigate 2 well known algorithms which utilize policy gradient methods, Proximal Policy Optimization (PPO) and Trust Region Policy Optimization (TRPO). To better demonstrate in practice how these algorithms would need to be implemented, we choose to utilize a simulation environment to train a quadrupedal robot to learn a gait planning and control policy. We demonstrate the use of developmental tools which help to expedite training, hyper-parameter tuning, and model analysis. Altogether we present a comparison of these results, and provide qualitative commentary on the behaviors learned by the quadrupedal robot over time between the two algorithms.

1 Introduction

We propose a straightforward approach to implementing, evaluating and optimizing deep policy gradient algorithms through the use of open-source optimization libraries and high-fidelity simulation environments. Our work shows promising results and opens up new questions regarding the best practices that could be followed when implementing reinforcement learning to a new environment. Our results showcase the ability of an extremely high dimensional non-linear dynamic system to be controlled by a learned control policy in a streamlined manner. In future work we plan on implementing these control policies onto our own physical system inspired by Stanford Doggo [1], seen in figure 1.

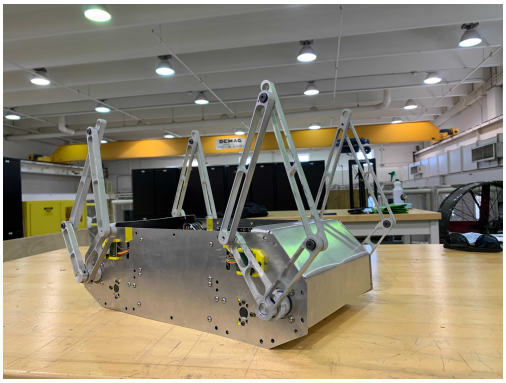


Figure 1: Clemson University Distributed Intelligence and Robot Autonomy (DIRA) Tigger Quadrupedal Robot [WIP].

*Masters Student, *Department of Mechanical Engineering*, akrolic@clemson.edu
†PhD Student, *Department of Mechanical Engineering*, ssutava@clemson.edu

1.1 Relevance and Applications

Robots and locomotion are an extremely popular and growing field of research and practice in the past decade. Open source projects such as the Stanford Doggo and Stotch which lower the cost barriers to study the control of these complex dynamical systems [1], [2]. Boston Dynamics has impressed the world with their vast array of monopeds, bipeds, and most popular, quadrupedal autonomous robots [3]. A key component of these modern marvels is their ability to adapt to the environment and execute elegant locomotion through even the most adverse conditions. Surely more than just a PID controller is going to have to cut it when it comes to approaching the solution to locomoting over uneven terrains in varying climates. Recent success in developing gait trajectory controllers and position controllers has emerged from solutions that utilize deep reinforcement learning techniques.

1.2 Existing Work

The focus of our research centers around quadrupedal robots, and recently researchers at ETH Robotics System Lab developed their own high fidelity simulation environment, Raisim, which is specifically designed for training reinforcement learning agents [4]. Access to this environment is only available to approved researchers, but benchmarks show [5] that it is superior to popular open source environments such as pybullet, MuJuCo, DART, et al [6]. Here are several papers which report high accuracy when compared to the physical machine [7], [8], [9]. Hierarchical deep reinforcement learning approaches have been explored for bidpedal locomotion [10] where planning and control are treated as two separate controllers which are learned together through reinforcement learning. This decoupling at the planning level is also seen in recent works on ANYMAL in the DeepGait papers [11], [12]. Although the decoupling of planning at the planning and control level has been seen before, there is interest in decoupling the control of the actuators themselves into multi-agent reinforcement learning problems where there is a shared objective that could be in the form of an shared observation or carefully crafted reward function. Some discussion around this topic is reserved for the simulation section since it deals with the specific application of quadrupedal robots.

1.3 Main Challenge

Most reinforcement learning algorithms are extremely sample inefficient, and often require millions of agent environment interactions before something useful can be learned. Decent results in reinforcement learning agents require algorithms to have their hyperparameters tuned. These hyperparameters could influence directly the policy, value function, rewards, and for gradient based methods the underlying deep neural networks hyperparameters such as the number of layers, hidden units, activation functions, ect. Its a complex optimization problem which is impossible to fully solve in a realistic setting, for example you are a student who has a report due in a week but training takes 24 hours for what could be a subpar result. To help alleviate some of this frustration, we recommend learning from our methods to ease the deployment of gradient based algorithms into any new complex environment.

2 Theory

2.1 Basics on policy gradient

A policy is a probability distribution over actions for an agent. Deep neural networks are a popular way to approximate the policy. The network takes the state as the input and gives out a distribution over actions. Policy gradient methods are the ones that do not rely on the value functions to get a policy. The objective that is being optimized is the expected discounted return concerning the policy given by eqn(1).

$$J(\theta) = \mathcal{V}_{\pi_\theta} = \mathop{\mathbb{E}}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{\infty} \gamma^t r_t \right] \tag{1}$$

This expectation can be over a distribution of possible initial states. The performance index is parameterized with parameters θ. The parameter update is made in the direction of the gradient of the performance index J, scaled by some factor α or gradient ascent in other words. All PG methods follow this general scheme in eqn (2).

$$\theta_{t+1} = \theta_t + \alpha \nabla \hat{J}(\theta_t) \tag{2}$$

Though, the value functions are not required, they can be used additionally to reduce the variance and speed up the learning. The methods that learn some form of value function, along-side the policy are termed as the actor critic methods, where the policy represent actor and value function represents critic.

2.2 Advantages of Policy Gradient Methods

There are some key factors that make the policy gradient methods attractive.

- Many RL algorithms, like deep-Q learning, rely on learning the estimates of the value functions. The agent must identify how valuable each state is to select the action that leads to the most important states. But this assumes that it is possible to learn the action-value function accurately. In many cases, the value functions could be incredibly complex and challenging to learn on realistic time scales. In some cases, a working policy may be much simpler and, therefore, easier to approximate. The policy gradient agent can learn to overcome certain situations much more swiftly than value-based methods. i.e., policy gradient could be more efficient.
- Another point that makes the policy gradient techniques attractive is what if the optimal policy is deterministic. For simplistic environments with apparent deterministic policy, keeping some finite ϵ could make the agent explore the environment unnecessarily. For some complicated settings (or tasks), the optimal policy may very well be deterministic but possibly not very obvious. In theory, the exploration part ϵ can be decreased with time and making the policy eventually settle to a deterministic policy. But, the rate of ϵ decay is challenging to decide. The policy gradient, even though stochastic, can approach a deterministic policy over time. More profitable actions are picked more frequently, creating momentum towards the locally optimal, deterministic policy.
- Based on the nature of the problem, it could be significantly easier and efficient to learn the policy directly, instead of learning it through a value function. Additionally, the policy gradient methods can be more effective in dealing with continuous state and action spaces. And most importantly, it is easier to induct the domain knowledge to guide the policy search in comparison with learning the value function.

2.3 Policy Gradient Theorem

The objective of reinforcement learning is to maximize the agent's performance. The weights and biases of the neural networks (θ) parameterize The performance index J. The update rule for the parameters is $\theta_{new} = \theta_{old}+$ some rate α times the gradient of the performance metric ($\nabla J(\hat{\theta_{old}})$). According to the policy gradient method,

$$\nabla J(\theta) \propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla_\theta \pi(a|s,\theta) \tag{3}$$

This is equivalent to an expectation (over a stationary distribution following policy θ) and following some mathematical manipulation results in,

$$\nabla J(\theta) = \mathbb{E}_\pi \left[\sum_a \pi(a_t|s_t,\theta_t) q_\pi(s_t,a_t) \frac{\nabla_\theta \pi(a_t|s_t,\theta_t)}{\pi(a_t|s_t,\theta_t)} \right] \tag{4}$$

$$= \mathbb{E}_\pi \left[q_\pi(s_t,a_t) \frac{\nabla_\theta \pi(a_t|s_t,\theta_t)}{\pi(a_t|s_t,\theta_t)} \right] \tag{5}$$

$$= \mathbb{E}_\pi \left[G_t \frac{\nabla_\theta \pi(a_t|s_t,\theta_t)}{\pi(a_t|s_t,\theta_t)} \right] \tag{6}$$

Therefore the update rule becomes (this the update rule for the REINFORCE algorithm),

$$\theta_{t+1} = \theta_t + \alpha G_t \frac{\nabla_\theta \pi(a_t|s_t,\theta_t)}{\pi(a_t|s_t,\theta_t)}$$

The G_t term is the discounted future returns (actually observed) and the gradient of the policy gives the direction in the policy space that maximizes the chance of repeating the action A_t when in state S_t. This process raises the probability of picking the action with large expected future returns, and the opposite happens to the actions with below-par expected future returns. The agent can learn to maximize the reward over time, making the policy gradient methods very powerful. A baseline could be included in the update term alongside the discounted future returns. Addition of the baseline does not add any bias but it can help with the variance. The update rule with baseline included is called as REINFORCE with baseline.

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha \left(G_t - b\left(S_t\right) \right) \frac{\nabla \pi \left(a_t \mid s_t, \boldsymbol{\theta}_t \right)}{\pi \left(a_t \mid s_t, \boldsymbol{\theta}_t \right)} \quad \textit{(REINFORCE with baseline)} \tag{7}$$

The term A_θ representing the difference between the Q-function and the value function V; where Q gives the expected return and V provides the baseline. A_θ is also called as the advantage estimate, because it tells us, how advantageous it

is, to choose an action compared to a weighted average over all actions. Gradient of the performance index in terms of advantage estimate is given in equation (9).

$$A_{\pi_\theta}(s,a) = Q_{\pi_\theta}(s,a) - V_{\pi_\theta}(s) \tag{8}$$

$$\nabla_\theta J(\theta) = \mathop{\mathbb{E}}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{\infty}\gamma^t \nabla_\theta \log \pi_\theta(a_t \mid s_t)\, A_{\pi_\theta}(s_t,a_t)\right] \tag{9}$$

2.4 Limitations of plain policy gradient

Policy gradient methods try to optimize the performance index we are interested in directly. But these methods are not very efficient in using old data (trajectories based on old/different policies) due to on-policy learning. Plain policy gradient substitutes the gradient estimate into the gradient ascent method. We can modify this method to more effectively update the policy based on the experience. Usually, the data is discarded after the computation of a single gradient. As a result, the information extracted in one iteration may only be a fraction of what is available for learning. This is even more prominent when the gradients vary over a large scale. Progress in the direction of small gradients will be slow and visa-versa. Instead of taking a gradient for every trajectory (such high-frequency changes make learning slow or even unstable), an optimization problem can be solved for a batch of data to find the most promising gradient direction. The agent has a significant probability of selecting any action in any given state; therefore, it is challenging to regulate the variation in the episodes due to the extremely large (infinite, in continuous case) number of possible actions.

It is difficult to obtain a learning rate α, that allows sufficient learning and at the same time keeps the algorithm stable. This is difficult to achieve, because estimating the size of the jump in policy space based on the update in the parameter space is not straightforward. It is challenging to choose a step size that works over the complete optimization process, allowing for a reasonable learning rate without inducing instability. The data that the neural network receives is non-stationary since as the policy changes, the data (observations and rewards) is sampled from a different distribution. Statistics of the input are changing over time, making it hard to choose a good step size that can work over the entire course of the learning process. A too big step is much more damaging in the RL problem than static optimization problems like supervised learning. Suppose we start collecting trajectories at any stage of the process from a bad policy that does not visit the optimal locations in the state-space. In that case, it becomes really difficult for the algorithm to come back to a more suitable policy after that point. Result in a steep drop in performance. It is challenging to design an algorithm that performs well and guarantees convergence to a local optimum.

The simple policy gradient methods can be improved upon by approximately solving an optimization problem using the data collected under current policy. (Vanilla policy gradient methods use the estimates/approximations of the policy gradients (these are obtained as the sample averages) to directly modify/update the parameters of the policy). The idea is that this helps in extraction of a bit more information from data compared to the plain approach. To deal with the variance between the episodes, we can scale the rewards with some baseline without introducing any bias. The simplest baseline to use is the average reward from the episodes. To deal with sample inefficiency, we can let the agent generate a batch of trajectories so that it has a chance of visiting the state more than once before updating the parameters. This introduces an additional hyper-parameter (batch size of updates) but usually results in significantly faster convergence to a good policy. Increasing the batch size could be the difference between no learning (at least visible in terms of performance) and something (some policy) that works nice.

The problem of sample inefficiency can be addressed to some extent by introducing the importance sampling for calculating the gradient as in equation (10). This is nothing but a consequence of the policy gradient theorem.

$$\nabla_\theta J(\theta) = \mathop{\mathbb{E}}_{\tau\sim\theta'}\left[\sum_{t=0}^{\infty}\frac{P(\tau_t \mid \theta)}{P(\tau_t \mid \theta')}\gamma^t \nabla_\theta \log \pi_\theta(a_t \mid s_t)\, A_\theta(s_t,a_t)\right] \tag{10}$$

$$\frac{P(\tau_t \mid \theta)}{P(\tau_t \mid \theta')} = \frac{\mu(s_0)\prod_{t'=0}^{t} P(s_{t'+1} \mid s_{t'},a_{t'})\,\pi_\theta(a_{t'} \mid s_{t'})}{\mu(s_0)\prod_{t'=0}^{t} P(s_{t'+1} \mid s_{t'},a_{t'})\,\pi_{\theta'}(a_{t'} \mid s_{t'})} = \prod_{t'=0}^{t}\frac{\pi_\theta(a_{t'} \mid s_{t'})}{\pi_{\theta'}(a_{t'} \mid s_{t'})} \qquad \text{(Importance sampling ratio)} \tag{11}$$

But the importance sampling itself may give rise to the issue of the exploding or vanishing gradients. As we can see from equation (11), the importance sampling ratio is a product of multiple fractions. These fractions are nothing but the probabilities of selecting the same action when in the same state, under two different policies. The fractions are calculated and multiplied over the entire trajectories. If there is significant mismatch between probabilities of the policies, for instance one of the probabilities in the denominator is close to zero but the numerator is significantly higher

or visa-versa then the value of the importance sampling ratio may blow up to a large number or die down to near zero. This makes the update unstable. Therefore, this measure is not enough to address the issues effectively and an alternate solution is still required. In this search for robust alternatives, two results turn out to be particularly helpful. The first one is on comparison of performance of two policies, and the second one is on the monotonic improvement in policy.

2.5 Comparing two policies

The difference in the performance of two policies π and π', as shown in equation (12), is equal to the discounted sums of advantage estimates of the policy π averaged over all possible trajectories generated using policy π'.

$$J(\pi') - J(\pi) = \mathop{\mathbb{E}}_{\tau \sim \pi'} \left[\sum_{t=0}^{\infty} \gamma^t A^{\pi}(s_t, a_t) \right] \tag{12}$$

$$= \frac{1}{1-\gamma} \mathop{\mathbb{E}}_{\substack{s \sim d^{\pi} \\ a \sim \pi'}} \left[A^{\pi}(s, a) \right] \tag{13}$$

$$= \frac{1}{1-\gamma} \mathop{\mathbb{E}}_{\substack{s \sim d^{\pi'} \\ a \sim \pi}} \left[\frac{\pi'(a \mid s)}{\pi(a \mid s)} A^{\pi}(s, a) \right] \tag{14}$$

$$\approx \frac{1}{1-\gamma} \mathop{\mathbb{E}}_{\substack{s \sim d^{\pi} \\ a \sim \pi}} \left[\frac{\pi'(a \mid s)}{\pi(a \mid s)} A^{\pi}(s, a) \right] \tag{15}$$

$$= \mathop{\mathbb{E}}_{\tau \sim \pi} \left[\sum_{t=0}^{\infty} \gamma^{\mathrm{t}} \frac{\pi'(a_t \mid s_t)}{\pi(a_t \mid s_t)} A^{\pi}(s_t, a_t) \right] \doteq L_{\pi}(\pi') \tag{16}$$

$$d^{\pi}(s) = (1-\gamma) \sum_{t=0}^{\infty} \gamma^t P(s_t = s \mid \pi) \tag{17}$$

If γ is strictly less than 1, then this can be equivalently written as in equation (14). Notice here, that the actions are sampled using policy π, but the states are sampled over the discounted future state distribution for policy π'. We can see this distribution as the steady state distribution of policy π and discount factor γ.

If two policies π and π' are close enough, then the state distribution $d^{\pi'}$ can be replaced with d^{π} and the equation (15) gives us an approximation of the relative performance of the two policies. Equation (16) is defined as the loss function. One thing that still remains unclear is, how good the approximation in equation (16) is. As it turns, out the absolute error between the actual loss and the loss function in (16) is bounded by some constant C, times the square root of the average KL divergence over all states of two policies according two equation (18).

$$|J(\pi') - (J(\pi) + L_{\pi}(\pi'))| \leq C \sqrt{\mathop{\mathbb{E}}_{s \sim d^{\pi}} \left[D_{KL}(\pi' || \pi)[s] \right]} \tag{18}$$

$$D_{KL}(\pi' \| \pi)[s] = \sum_{a \in \mathcal{A}} \pi'(a \mid s) \log \frac{\pi'(a \mid s)}{\pi(a \mid s)} \tag{19}$$

The KL divergence term here is used as a metric of distance between two distributions. Since equation (16)is a good approximation of equation (12), for π and π' sufficiently close, advantage eqn(16)offers over (12) is that, the trajectories are sampled using policy π, but it somehow gives an estimate on how good the policy π' is. Therefore we can form an equivalent optimization problem, that can help us find the next best policy π' in the neighbourhood of π.

2.6 Monotonic Improvement in Policy

This optimization problem takes the form of equation (20), which uses maximum KL divergence between policies π and π' as the regularization term to represent the worst case scenario.

$$\max_{\pi'} L_{\pi_k}(\pi') - C \max_{s \sim d^{\pi_k}} \left[D_{KL}(\pi' \| \pi_k)[s] \right] \tag{20}$$

The monotonic improvement theory then guarantees that, the optimal new policy π_{k+1} will be at least as good as the old policy π_k. But, optimizing equation (20) is a difficult task, especially due to maximum over KL divergence term. Therefore an approximation of this is obtained as equation (21), where the objective is the loss function and a constraint

is included that limits the average KL divergence to be less that some small positive number δ. Equation (21) is the basis of TRPO and PPO.

$$\begin{aligned} &\arg\max_{\pi'} L_{\pi_k}(\pi') \\ &\text{s.t.} \underset{s\sim d^{\pi_k}}{E}\left[D_{KL}\left(\pi' \| \pi_k\right)[s]\right] \leq \delta \end{aligned} \tag{21}$$

2.7 Trust Region Policy Optimization

In particular for TRPO, the objective loss function is approximated as a linear/affine function of parameter θ, about the old loss function L_{θ_k}. The constraint is approximated as a quadratic function of θ. Therefore the approximated problem becomes a problem with linear objective and quadratic constraint, as in equation (24).

$$L_{\theta_k}(\theta) \approx L_{\theta_k}(\theta_k) + g^T(\theta - \theta_k) \quad g \doteq \nabla_\theta L_{\theta_k}(\theta)\big|_{\theta_k} \tag{22}$$

$$\bar{D}_{KL}(\theta \| \theta_k) \approx \frac{1}{2}(\theta - \theta_k)^T H(\theta - \theta_k) \quad H \doteq \nabla^2_\theta \bar{D}_{KL}(\theta \| \theta_k)\Big|_{\theta_k} \tag{23}$$

$$\begin{aligned} &\arg\max_{\theta} g^T(\theta - \theta_k) \\ &\text{s.t.} \ \frac{1}{2}(\theta - \theta_k)^T H(\theta - \theta_k) \leq \delta \end{aligned} \tag{24}$$

This formulation has an analytic solution given by equation (25).

$$\theta_{k+1} = \theta_k + \sqrt{\frac{2\delta}{g^T H^{-1} g}} H^{-1} g \tag{25}$$

But calculating H^{-1} for equation (25) is impractical for large systems. Therefore, $H^{-1}g$ is estimated using conjugate gradient method. Because of all these steps of approximation, there is no guarantee that $L_{\theta_k} \geq 0$ (which is required for improved next policy) or the KL divergence between policies is less than the threshold δ (which is required to make sure that the approximation holds). Therefore, a line search method is implemented in the direction of the estimated gradient, which tries to find the biggest step size parameter, by exponentially reducing the size of the steps, till the conditions are met.

2.8 Proximal Policy Optimization

The PPO tries to solve a different form of equation (20). In **PPO with adaptive KL penalty**,

$$\arg\max_{\theta} L_{\theta_k}(\theta) - \beta_k \bar{D}_{KL}(\theta \| \theta_k) \tag{26}$$

the regularizing term maximum over the KL divergence is replaced by average over KL divergence. β_k is updated after every every batch. $d = \bar{D}_{KL}(.)$, if $d \leq d_{targ}/1.5, \beta \leftarrow \beta/2$, if $d \geq d_{targ} * 1.5, \beta \leftarrow \beta * 2$. The problem can be approached via mini-batch stochastic gradient descent using Adam type optimizers. But in this algorithm the KL constraint is not always satisfied, also the effect of adaptive β_k is not as straight forward as effect of changing δ in TRPO. The empirical performance of this type of PPO is not as good as the PPO with Clipped Objective.

In **PPO with clipped objective**,

$$r_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\text{old}}(a_t \mid s_t)}, \qquad \text{clip}(r, a, b) = \begin{cases} a \text{ if } r < a \\ b \text{ if } r > b \\ r \text{ otherwise} \end{cases} \tag{27}$$

$$L^{CLIP}_{\theta_k}(\theta) = \underset{\tau\sim\pi_k}{\mathbb{E}}\left[\sum_{t=0}^{T}\left[\min\left(r_t(\theta)\hat{A}^{\pi_k}_t, \text{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}^{\pi_k}_t\right)\right]\right] \tag{28}$$

$$\arg\max_{\theta} L^{CLIP}_{\theta_k}(\theta) \qquad \text{(objective without critic)} \tag{29}$$

$$L_t^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t \left[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S\left[\pi_\theta\right](s_t) \right] \tag{30}$$

$$\arg\max_\theta L_{\theta_k}^{CLIP+VF+S}(\theta) \qquad \text{(objective with critic)} \tag{31}$$

the regularizing term is removed and the objective is modified to include a clipping term, expected to emulate the effect of the KL constraint. This clipping term does not allow the probability ratio $r_t(\theta)$ to go out of a predefined range. As a result the clipped objective function looks like the one in equation (29).

Unlike TRPO, both the PPO algorithms rquire a separate arrangement to estimate the advantages A_π. Therefore, these methods also fall under the actor-critic category. If DNN based parameterization is used, for PPO and the same network estimates both the policy and advantage $\hat{A}_\pi$, then the loss function is modified to include additional terms. L_t^{VF}: for reducing error in the estimation of the value function, and $s[\pi_\theta](s_t)$: entropy regularization term to make sure that enough exploration happens. The modified objective function is then expressed as in equation (31). This in effect allows the implementation of parallalizable actor-critic methods like A2C, for simulations.

3 Algorithm

Algorithm 1 Trust Region Policy Optimization [13]

Input: Initial policy parameters θ_0
Output: Optimal policy parameters $\theta^\star$
for $k = 0, 1, 2, \ldots$ **do**
  Collect set of trajectories $\mathcal{D}_k$ on policy $\pi_k = \pi\left(\theta_k\right)$
  Estimate advantages $\hat{A}_t^{\pi_k}$ using any advantage estimation algorithm
  Form sample estimates for
  - policy gradient $\hat{g}_k$ (using advantage estimates)
  - and KL-divergence Hessian-vector product function $f(v) = \hat{H}_k v$
  Use CG with n_{cg} iterations to obtain $x_k \approx \hat{H}_k^{-1} \hat{g}_k$
  Estimate proposed step $\Delta_k \approx \sqrt{\frac{2\delta}{x_k^T \hat{H}_k x_k}} x_k$
  Perform backtracking line search with exponential decay to obtain final update
    $\theta_{k+1} = \theta_k + \alpha^j \Delta_k$
 end for
return

Algorithm 2 Proximal Policy Optimization [13]

Input: Initial policy parameters θ_0
Output: Optimal policy parameters $\theta^\star$
for $k = 0, 1, 2, \ldots$ **do**
  Collect set of partial trajectories $\mathcal{D}_k$ on policy $\pi_k = \pi\left(\theta_k\right)$
  Estimate advantages $\hat{A}_t^{\pi_k}$ using any advantage estimation algorithm
  Compute policy update
    $\theta_{k+1} = \arg\max_\theta \mathcal{L}_{\theta_k}^{CLIP}(\theta)$
  by taking K steps of minibatch SGD (via Adam), where
    $\mathcal{L}_{\theta_k}^{CLIP}(\theta) = \mathop{\mathrm{E}}_{\tau \sim \pi_k} \left[\sum_{t=0}^{T} \left[\min\left(r_t(\theta) \hat{A}_t^{\pi_k}, \mathrm{clip}\left(r_t(\theta), 1-\epsilon, 1+\epsilon\right) \hat{A}_t^{\pi_k} \right) \right] \right]$
 end for

return

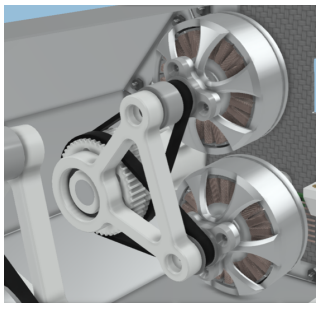
4 Results

In reinforcement learning, agents train on data which is collected through their interactions with the environment. Typically, the algorithms are sample inefficient and require extended periods of time to collect the interactions between

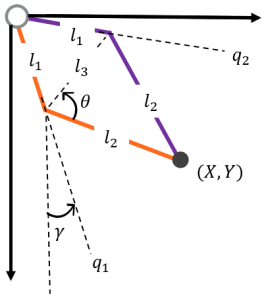
the agent and environment. Its for this reason that selecting a high fidelity simulation environment to generate the training data is an important requirement, ensuring that the simulation represents the physics of the problem being investigated. We choose to use the Bullet Physics Engine by creating a PyBullet [6] simulation environment to train a control policy for a quadrupedal robot. We can also utilize the open source openai gym library to help streamline implementation in a consice fashion [14] The decision to use this simulation environment was based on the past success of researchers [15] in transferring their learned policies from the simulation environment onto the physical robot. This 'reality gap' between the simulated environment and real world is often a limiting factor to most simulation environments that fail to capture an accurate model of the actuators dynamics and neglect the latency that exists in a real robotic system.

4.1 Simulation:

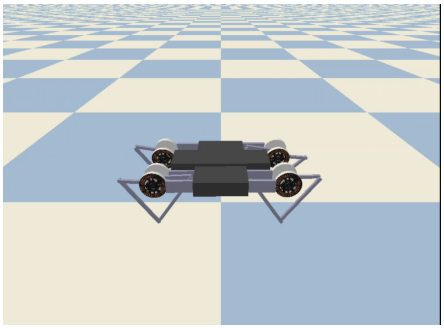
The quadrupedal robot model used in the PyBullet simulation is the Minitaur from Ghost Robotics as seen in figure 2c. The robot's legs use two motors directly connected to the legs upper linkages. A similar actuator model is shown in Figure 2a where there is a gear train in between the motor and linkages. The motors are aligned co-axially and are actuated through position control sequences, the legs coordinate of reference is shown in figure 2b. The model of the robot contains information about the mass of the components, friction at the joints, and integrates a more accurate actuator model.



(a) Quasi-Direct Drive Actuators



(b) Symmetric five bar linkage



(c) Minitaur Agent

Figure 2: Quadruped Agent and Actuator Model

An important first step to formulating the reinforcement learning problem within this environment is defining the observations, actions, and the reward function. The observation space $S \in \mathbb{R}^{28}$ for the agent is based on proprioceptive measurements from the actuators and pose measurements from at the center of mass of the robot. Altogether the observation space includes pose, p, quaternions. The observations extend to the actuator, where each legs 2 motors provide their motor angles q, motor angular velocities, $\dot{q}$, and motor torques T. For our quadruped robot this totals to 8 motors with 6 observable states per leg. In summary

$$\begin{aligned} \text{p} \in S &= \vec{x}, \vec{y}, \vec{z}, \vec{w} \\ q \in S &= q_{11}, q_{12}, q_{21}, q_{22}, q_{31}, q_{32}, q_{41}, q_{42} \\ \dot{q} \in S &= \dot{q}_{11}, \dot{q}_{12}, \dot{q}_{21}, \dot{q}_{22}, \dot{q}_{31}, \dot{q}_{32}, \dot{q}_{41}, \dot{q}_{42} \\ T \in S &= T_{11}, T_{12}, T_{21}, T_{22}, T_{31}, T_{32}, T_{41}, T_{42} \end{aligned} \tag{32}$$

The reward function is designed to maximize linear forward velocity and penalize effort from the actuators. The reward function is defined explicitly in equation (33) below.

$$R = (p_n - p_{n-1})\ \vec{x}\ \ -\ w\Delta t\,|T_n\dot{q}_n| \tag{33}$$

Where p_n is the position of the center of mass in the current time step and p_{n+1} is the position of the center of mass in the next time step. The actuator model developed for this simulation environment models the dynamics of an ideal DC motor. This actuator model was validated against the physical robot in the real world, and it was determined to be accurate. Another possible approach to this could be through means of system identification, where the actuator is modeled as a neural network. This approach may be advantageous for quasi-direct drive or linear actuator quadrupedal leg designs, but since in this case the motors are directly linked to the legs without any intermediary components, it is easier to take the direct approach as was done here.
The environment resets once the terminal state is reached. The terminal state is defined either to be the 1000th time step, or when the center of mass height falls below 0.13 meters with respect to the ground. These 2 termination conditions ensure that if the agent falls, the environment will reset. If the agent is able to continue to walk for the 1000 forward steps of the simulation, then the robot will return to its initial position and begin again. The logic here is that you want to avoid learning behaviors that might emerge from potential exploitation of the environment since physical damage is not factored into falling over.

4.2 Hyperparameter Tuning

The algorithms used in our research were implemented by stable-baselines as an improvement to the original implementation from openai's baselines algorithm library. Each algorithms policy and value function is approximated using a deep neural network.The Policy Network, π_θ, has 2 layers with 64 hidden units each. Similarly, the Value Function Network, V_{π_θ}, has 2 layers with 64 hidden units each. We did not tune the network parameters, instead we ran permutations of different parameters which are directly linked to the algorithms being implemented. We used a open source hyperparameter optimization library, optuna, to perform a study over the range of hyperparameter values given in Table 1 [16].

Optimal Hyperparameters for 'MinitaurBulletEnv-v0'		
Hyperparameter	TRPO	PPO2
Number of State Steps until Terminal State	16 : 2048	16 : 2048
Kullback-Leibler loss threshold	1e-5 : 1	-
The compute gradient dampening factor	1e-3 : 0.1	-
Value Function Step Size	3e-6 : 3e-3	-
Discount Factor	0.9 : 0.9999	0.9 : 0.9999
Learning Rate	-	1e-5 : 1
Entropy Coefficient	1e-8 : 1e-1	1e-8 : 1e-1
Clipping parameter controlling policy update rate	-	0.1 : 0.4
Num. of epochs when optimizing the surrogate objective function	-	1 : 48
Generalized Advantage Estimator factor	0.8 : 1	0.8 : 1

Table 1: Range of hyperparameters for optuna optimization TRPO and PPO2.

The hyperparameter optimization process took 24 hours to complete for each algorithm. There were 100 trails through which the optimization algorithm tuned the hyperparameters to obtain the agent with the best average reward after training for 200,000 episodes. Classical approaches to hyperparameter tuning include brute force search approaches where a full factorial or latin hypercube sampling of the parameter space is sampled in order to obtain the best final parameterization. The optimization algorithm performs convex optimization using the log likelihood estimate of the function being optimized. In this case, our function is our reinforcement agent and instead of find the set of hyperparameters that minimizes our reward, we need to make the rewards earned during the evaluation of the studied hyperparameter set negative. This way the optimizer is minimizing the negative log likelihood in order to actually perform the maximum likelihood estimate of the algorithms hyperparameter set. Optuna uses the scikit optimization library skopt in order to choose the best next values of hyper-parameters to tune to maximize the agents reward.

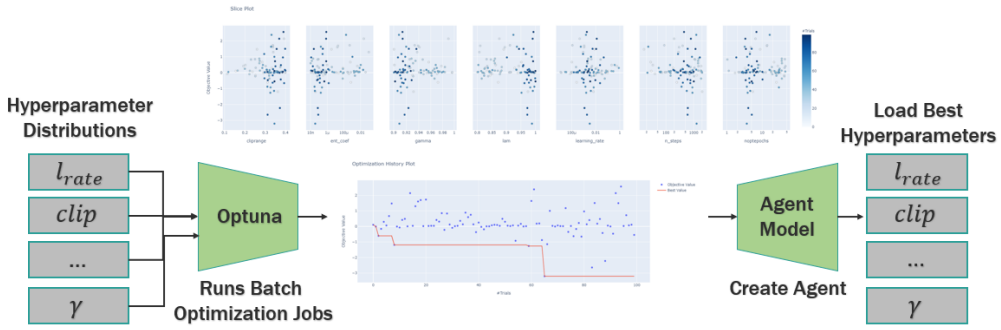


Figure 3: Hyperparameter optimization pipeline

The results for TRPO are shown in figure 4 the dots show the objective value from the optimizer minimizing the negative reward value for the given trial of hyperparameters. The red line shows the best value obtained during the optimization, and in this case only about 10 percent of all trials obtained a positive reward. It is possible that a better performing set of hyperparameters could have been found but performed poorly during the evaluation stage of the process. This kind of "dumb luck" is what makes finding the best set of hyperparameters extremely difficult for increasing non-linear dynamic systems.

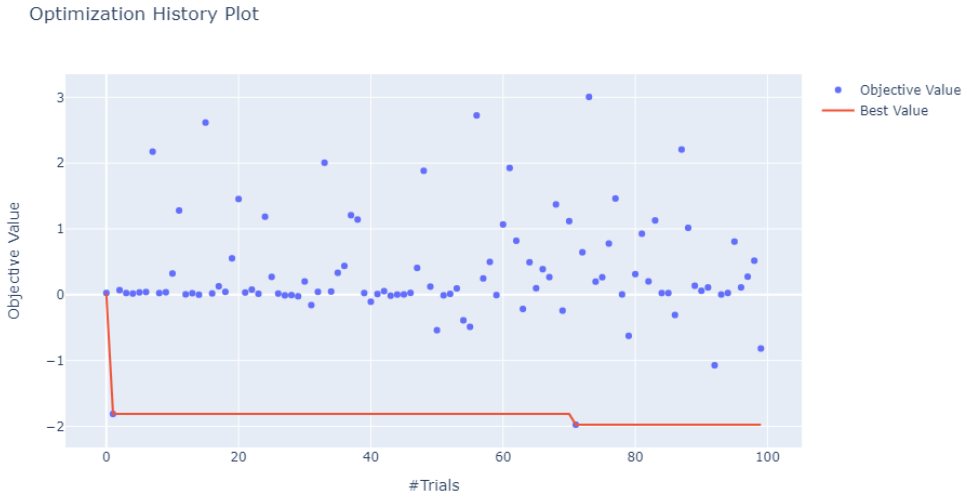


Figure 4: TRPO optimization coverage maximizing the skopt objective function.

In figure 5 the hyperparameters tuning range is plotted and the points show the value of the parameter, the corresponding objective value and the darker the dots indicate a larger number of trials evaluated at that particular operating point. graphs that show low variance in the sample space indicate that the optimal parameter value is likely somewhere within the neighborhood if the mean. A more robust testing strategy would be to take these results and refine the range of values to be optimized over within 2 standard deviations of the mean value shown in these graphs. A more in depth study of the sensitivity of certain values to change can also provide insight as to the extent of which this process should be repeated before accepting the final parameterization.

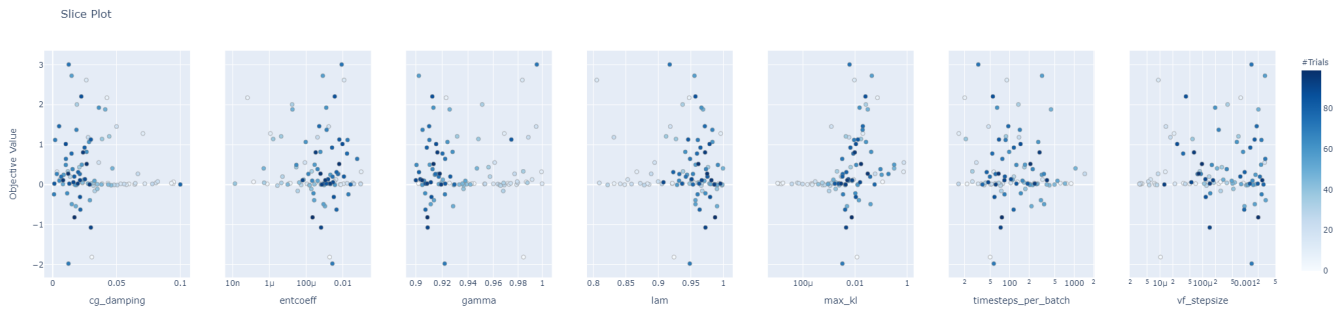


Figure 5: TRPO optimization hyperparameter accuracy per number of trials.

Similar to figure 4, in figure 6 PPO2 followed the same process and achieved generally more successful runs nearing 22 percent positive rewards. Ideally, this objective value would converge more elegantly to some large negative value, but for both algorithms it may be that the training was not long enough during optimization to truly explore the parameter space. Longer training times and a greater number of trials should in theory eventually obtain a excellent set of hyperparameters to maximize the defined reward function. Including hyperparameters to tune the reward functions weights during this process could be a great way to tune more than just the learning algorithm, but this was beyond the scope of this study.

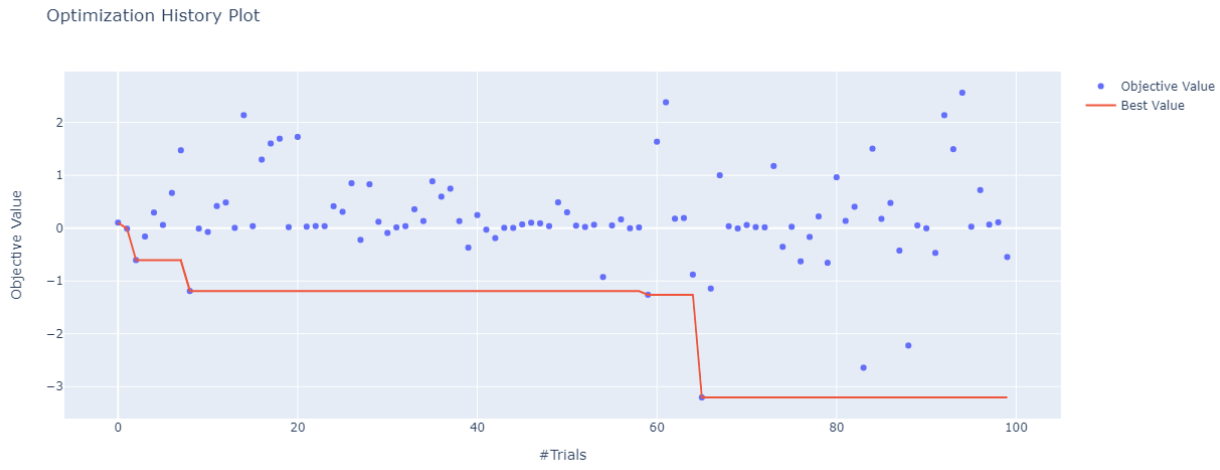


Figure 6: PPO2 optimization coverage maximizing the skopt objective function.

In figure 7, we can see where PPO2 performed best for each of the variable hyperparameters studied during optimization. In both cases for PPO2 and TRPO these plots indicate that the number of trials was sufficient to obtain a reasonably accurate final solution prior to moving onto the final long term training process.

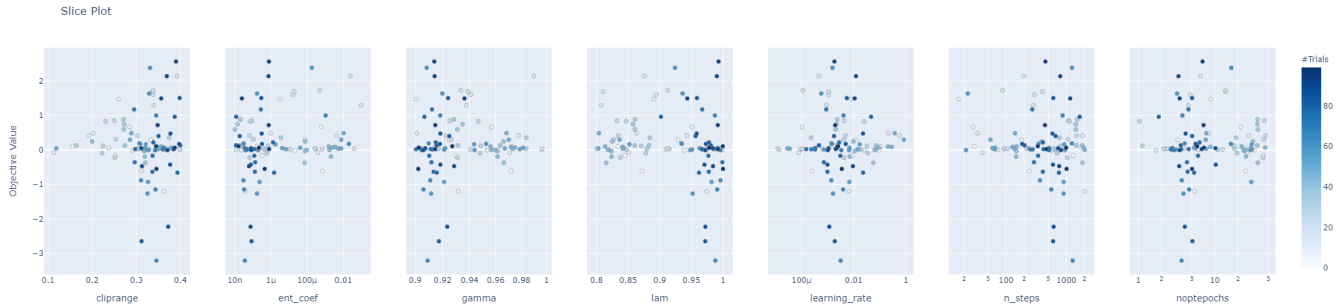


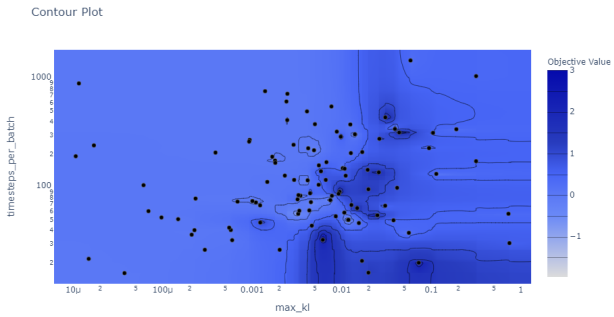
Figure 7: PPO2 optimization hyperparameter accuracy per number of trials.

Altogether the shared hyperparameters between the two algorithms such as the discount factor 'gamma', the generalized advantage estimator factor 'lam', the number of steps to termination 'n steps'/'timesteps per batch' were trained within the same starting range. The discount factors and generalized advantage estimator factors between the two algorithm converges to the same value, however the number of episodes to termination and entropy coefficient are several orders of magnitude different. Overall PPO2 was able to find a better initial reward during the 200,000 training episodes, but in similar work [15] only began to see real results around 4 to 7 million episodes before converging. Their average reward was sitting around 3-4 for trotting and 6-7 for galloping gaits learning from a reference trajectory. Immediately from scratch with the optimal hyperparameters for each algorithm we can see rewards as high as 2-3 which is extremely promising. A summary of the optimization results is shown in table 2.

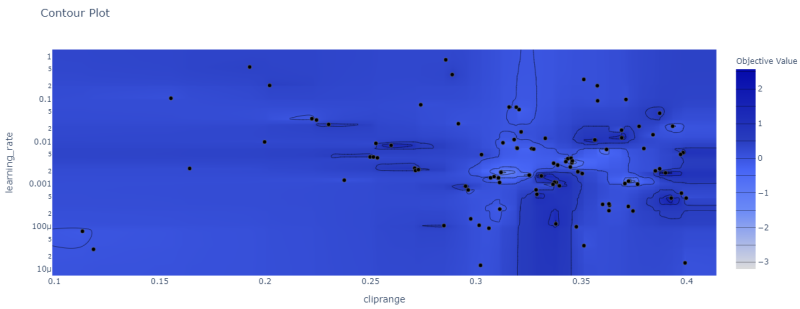
Optimal Hyperparameters for 'MinitaurBulletEnv-v0'		
Hyperparameter	TRPO	PPO2
Number of State Steps until Terminal State	293	1276
Kullback-Leibler loss threshold	0.0503	-
The compute gradient dampening factor	1.35e-2	-
Value Function Step Size	3.2e-3	-
Discount Factor	0.974	0.909
Learning Rate	-	3.17e-3
Entropy Coefficient	5.03e-3	3.59e-8
Clipping parameter controlling policy update rate	-	0.345
Num. of epochs when optimizing the surrogate objective function	-	4
Generalized Advantage Estimator factor	0.988	0.988

Table 2: Optimal hyperparameters for TRPO and PPO2 after 100 trails and 200,000 training steps per trial.

One additional advantage that makes using this optimization pipeline great for new environments is that you can visualize the trade-off between hyperparameters to develop better intuition as to how your algorithm fits to the given problem. This is shown for completetion with examples for PPO2 in figure 8a and TRPO in figure 8b.



(a) TRPO time to termination and KL Loss Threshold



(b) PPO2 learning rate and clip

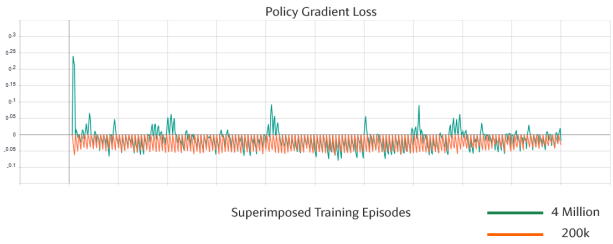
Figure 8: Topological parameter trade-off

4.3 Trained Agents

PPO2 Agent
Using the optimal hyperparameters from table 2, an agent was trained for 4 Million training iterations. Benchmark models were saved every 200,000 episodes to later on be able to load the models and evaluate their behavior as training progressed. The figures in 9 show a superimposed plot of the initial 200,000 episodes and the final 4,000,000 episodes episodic reward, value function loss and policy gradient loss. These 3 metrics alone can give a good idea as to the convergence of the agent with respect to the local optimum it found at a consistent 0.5 ± 0.05 average episodic reward. Just based on the data its clear that the agent learned this behavior near towards the end of the first 200,000 episodes and never was able to improve its behavior to achieve a different reward. So despite having the apparent best optimal hyperparameters for the environment, it appears that a cautious solution to exploit the more stable consistent return was the ultimate result. A video showing the final behavior of the PPO agent can be found here:

(a) Episodic Reward Start to Finish

(b) Value Function Loss

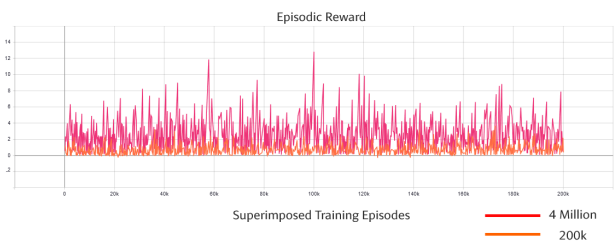


(c) Policy Gradient Loss

Figure 9: PPO2 Agent Training Results Superimposed at 200k and 4M training episodes

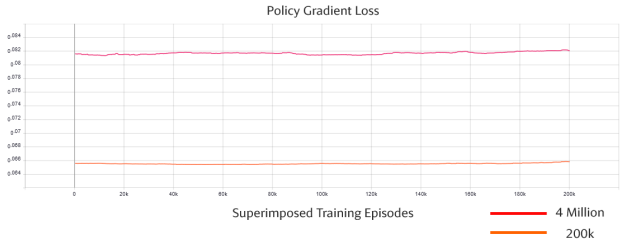
TRPO Agent
The same plots shown for PPO2 are shown for TRPO in figure 10. Immediately there is a stark contrast in the variance of the reward signal, and its increasing as the agent improves towards 4 Million episodes. This is a direct consequence of the learning approach, finding a stable policy to perform a fast gait motion satisfying only the simple goals defined

in the reward function does not guarantee elegance in the learned result. Every miss step or mistake due to poor foot placement will immediately impact the reward signal. The average reward for the trained TRPO agent was 2.5 ± 2. A video showing the learning of the TRPO agent from 200,000 episodes to 3,000,000 episodes can be found here: `https://youtu.be/Z4BZgIL5d0Y`



(a) Episodic Reward Start to Finish

(b) Value Function Loss



(c) Policy Gradient Loss

Figure 10: TRPO Agent Training Results Superimposed at 200k and 4M training episodes

An ideal agent for this environment would be one that combines the early success of TRPO to learn a fast gait motion, and PPO to help stabilize the learned policy and value function networks. This idea of a combined learning approach is what inspired the results obtained in the following paragraph.

Combined Approach

The advantage of keeping the size of the networks the same between the TRPO and PPO2 agents is that it is relatively easy to transfer the weights and biases of the policy and value function networks from one algorithm to another. This process is visualized in 11. After the network parameters have been manually transferred into the PPO2 model, the hyperparameter optimization scheme outline in figure 3 was performed to maximized the likelihood of improving the current policy and obtaining a more stable control policy.

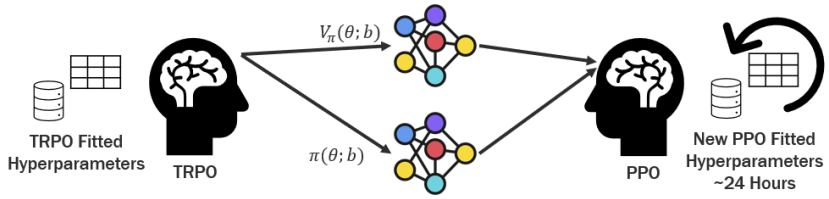


Figure 11: The transfer of a policy and value function network from TRPO to PPO2 [TRPO + PPO2 = PPO2*]

When looking at the results of the hyperparameter optimization in table 3 it can be seen that some parameters stayed the same from the initial optimization of a untrained policy and value function networks. Specifically the number of state steps until terminal state and discount factor remained about the same. The learning rate decreased by 2 orders of magnitude which is expected since the network is already performing well in terms of episodic rewards, so its best to make small changes in the network during backpropogation. The number of epochs when optimizing the surrogate objective function is lower and at the lowest bound of the parameter optimization range. This makes sense because we want continue to make small improvements to the policy during the gradient ascent of the policy networks loss. The

clipping parameter controlling the policy update rate also follows the same trend, as it is greater than the previous initial value for the untrained PPO2 agent. Higher clipping values lead to smaller changes in the policy, requiring even more stringent constraints on what is learned and what is thrown away during optimization of the surrogate policy. A lower Generalized advantage estimator factor also makes it more difficult to update the network since the loss function for the clipped objective function is minimizing between the new and old policies. A smaller entropy coefficient helps to reduce the number of random actions selected during training and should result in a more refined policy.

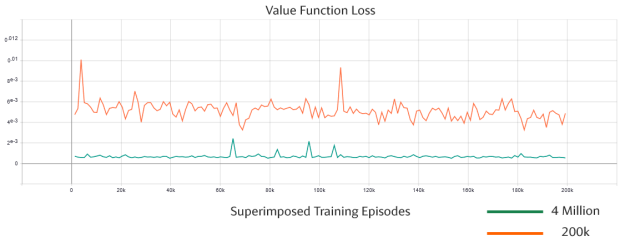
Comparing New PPO2* and Old Optimal Hyperparameters for 'MinitaurBulletEnv-v0'		
Hyperparameter	PPO2	PPO2*
Number of State Steps until Terminal State	1276	1277
Discount Factor	0.909	0.913
Learning Rate	3.17e-3	1.83e-5
Entropy Coefficient	3.59e-8	5.0e-4
Clipping parameter controlling policy update rate	0.345	0.379
Num. of epochs when optimizing the surrogate objective function	4	1
Generalized Advantage Estimator factor	0.988	0.861

Table 3: Optimal hyperparameters for PPO2 and PPO2*. Note PPO2* is not learning from scratch

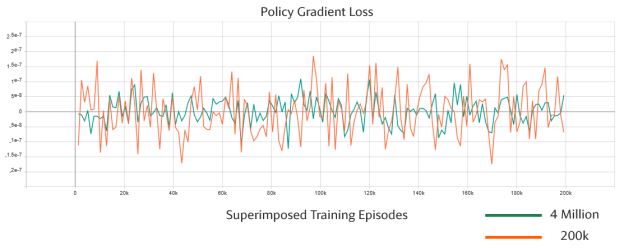
PPO2* Agent

Following the discussion from the previous paragraph, we have some expectations about what we would like to see from the PPO2* agent. The training results are shown in figure 12, and immediately we can observe that the agent is able to achieve an average episodic reward of 3 ± 1.5 which is a significant improvement. Looking more closely at the performance of the value function network we can see that from the initial 200,000 loss values the agent was able to reduce the overtime with less variance overtime. The value function is not nearly as precise as what we observed from TRPO in figure 10b but given the trend, this agent may need more time to train in order to converge to the optimal value function. The biggest improvement is actually in the policy gradient loss when compared to the TRPO in figure 10c where the PPO2* agent is able to even reduce the variance if you look closely into figure 12c. Altogether this agent is an improved iteration on the policy and value function networks learned by TRPO. Further work is merited in the best strategies for learning with multiple agents at different points of the training.

(a) Episodic Reward Start to Finish



(b) Value Function Loss



(c) Policy Gradient Loss

Figure 12: PPO* Agent Training Results Superimposed at 200k and 4M training episodes

4.4 Summary

The high fidelity physics environment and actuator models lay the framework for transferable policies from simulation onto a real quadrupedal robot. We learned the importance of hyperparameter tuning when initiating the training process for a new algorithm on a previously untested agent and environment. Utilizing Optuna to optimize the hyperparameter space is a cost effective and easy to implement solution when utilizing a gym structured environment for training and evaluation of the simulated agent. Hyperparameter optimization for this project consisted of optimizing 3 algorithms each over 100 trials where the agent is trained 200,000 episodes per trial on a studied set of hyperparameters. Each agents total hyperparameter optimization time took 24 hours to complete, which could be reduced in future implementation by taking advantage of newer implementations of the stable-baselines algorithms where parallelization is extended to a greater number of algorithms. We demonstrated that it is possible to transfer the policy and value function networks of one algorithm to another with relatively little difficulty.
When comparing the relative performance of the PPO2, TRPO, and PPO2* agents we can look to figure 13 where the black line is a moving average for the noisy reward signal. Initially, PPO2 converged nicely to a suboptimal local minimum policy and never explored beyond the initial policy space. TRPO however, was able to explore early on and near 1 Million episodes it begins to peak. It wavers slightly over the remaining training episodes and eventually is able to reduce some of the initial variance in the reward signal. PPO2* has a similar trajectory to the initial PPO2 but does not do well enough to minimize the noise in the reward signal and learn a more stable policy in the given training range. After closer inspection of the value function loss in the previous section, it was hypothesized that much longer training would be necessary for the convergence of the value function network. Overall PPO2* provides the best results but its still desirable to obtain a more stable control policy.

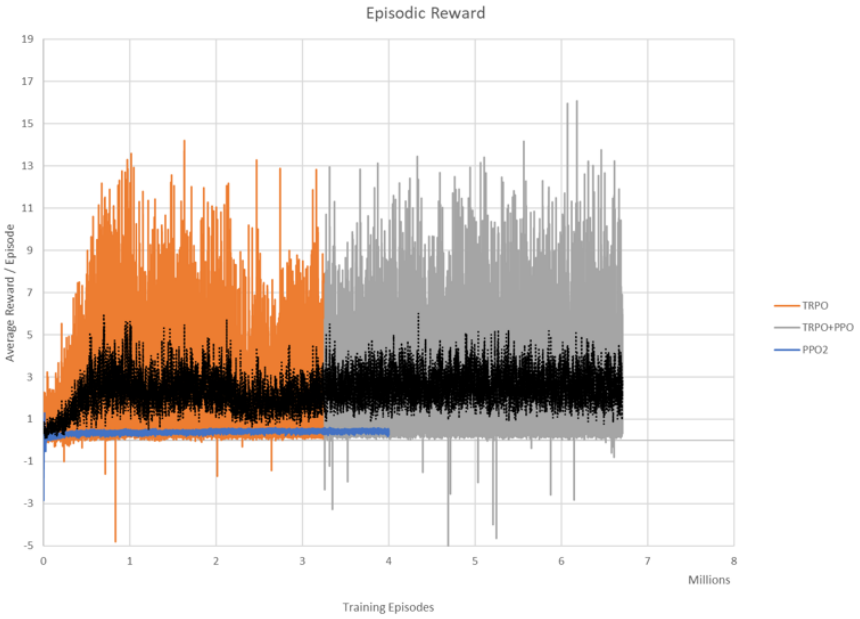


Figure 13: Average Reward/1000 episodes for all 3 Agents

References

[1] Nathan Kau, Aaron Schultz, Natalie Ferrante, and Patrick Slade. Stanford doggo: An open-source, quasi-direct-drive quadruped, 2019.

[2] Abhik Singla, Shounak Bhattacharya, Dhaivat Dholakiya, Shalabh Bhatnagar, Ashitava Ghosal, Bharadwaj Amrutur, and Shishir Kolathaya. Realizing learned quadruped locomotion behaviors through kinematic motion primitives, 2019.

[3] Boston Dynamics. "spot: Boston dynamics." spot | boston dynamics. `www.bostondynamics.com/spot`.

[4] Jemin Hwangbo, Joonho Lee, and Marco Hutter. Per-contact iteration method for solving contact dynamics. *IEEE Robotics and Automation Letters*, 3(2):895–902, 2018.

[5] Dongho Kang and Jemin Hwangho. Simbenchmark is a benchmark suite for state-of-the-art physics engines. `https://github.com/leggedrobotics/SimBenchmark`.

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

[7] Jemin Hwangbo, Joonho Lee, Alexey Dosovitskiy, Dario Bellicoso, Vassilios Tsounis, Vladlen Koltun, and Marco Hutter. Learning agile and dynamic motor skills for legged robots. *Science Robotics*, 4(26), 2019.

[8] Fan Shi, Timon Homberger, Joonho Lee, Takahiro Miki, Moju Zhao, Farbod Farshidian, Kei Okada, Masayuki Inaba, and Marco Hutter. Circus anymal: A quadruped learning dexterous manipulation with its limbs, 2020.

[9] Joonho Lee, Jemin Hwangbo, Lorenz Wellhausen, Vladlen Koltun, and Marco Hutter. Learning quadrupedal locomotion over challenging terrain. *Science Robotics*, 5(47), 2020.

[10] Xue Bin Peng, Glen Berseth, Kangkang Yin, and Michiel Van De Panne. Deeploco: Dynamic locomotion skills using hierarchical deep reinforcement learning. *ACM Trans. Graph.*, 36(4), July 2017.

[11] Vassilios Tsounis, Mitja Alge, Joonho Lee, Farbod Farshidian, and Marco Hutter. Deepgait: Planning and control of quadrupedal gaits using deep reinforcement learning, 2020.

[12] Jemin Hwangbo, Joonho Lee, Alexey Dosovitskiy, Dario Bellicoso, Vassilios Tsounis, Vladlen Koltun, and Marco Hutter. Learning agile and dynamic motor skills for legged robots. *Science Robotics*, 4(26):eaau5872, Jan 2019.

[13] Joshua Achiam. Advanced policy gradient methods. *Presentation*, 2017.

[14] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[15] 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. *CoRR*, abs/1804.10332, 2018.

[16] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. Optuna: A next-generation hyperparameter optimization framework. *CoRR*, abs/1907.10902, 2019.