Interpretable Reinforcement Learning for Continuous Action Environments

Extending DTPO for Continuous Action Spaces and Evaluating Competitiveness with RPO

Misha Kaptein

Supervisors: Anna Lukina, Daniël Vos

[1]**EEMCS, Delft University of Technology, The Netherlands**

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering
June 19, 2025

Name of the student: Misha Kaptein
Final project course: CSE3000 Research Project
Thesis committee: Anna Lukina, Daniël Vos, Luciano Cavalcante Siebert

Abstract

This research addresses the challenge of interpretability in Reinforcement Learning (RL) for environments with continuous action spaces by extending the Decision Tree Policy Optimization (DTPO) algorithm, which was originally developed for discrete action spaces. Unlike deep RL methods such as Proximal Policy Optimization (PPO), which are effective but difficult to interpret, DTPO offers transparent rule-based policies. We propose a continuous-action variant of the DTPO algorithm, DTPO-c, which allows decision trees to output Gaussian distribution parameters while maintaining interpretability. Our experiments on the Pendulum-v1 environment show that DTPO-c can achieve performance comparable to Robust Policy Optimization (RPO), although it requires more computational effort. Additionally, we investigate the impact of discretizing continuous actions and find that increasing action resolution does not always lead to improved performance, likely due to limited model capacity. These results confirm the feasibility of interpretable RL in continuous environments, making it suitable for applications where understanding and trusting the behavior of the model is important.

1 Introduction

Reinforcement Learning (RL) is a field within machine learning in which agents are trained to learn to make decisions in uncertain sequential environments by interacting with the environment. For every timestep, the agent observes the current state of the environment, selects an action, receives a reward, and transitions to a new state. Over time, the agent learns to maximize its total reward or minimize the total penalty by balancing exploration with exploitation [1]. In recent years, RL has been increasingly used in areas such as healthcare and autonomous driving vehicles [2, 3].

The most used approach for solving RL problems in complex environments is by using deep reinforcement learning, where deep neural networks are used to represent policies and value functions. Algorithms such as Deep Q-Networks (DQN) [4] and Proximal Policy Optimization (PPO) [5] have shown strong performance on a variety of tasks. However, a problem is that these models are mostly black boxes, where we do not completely understand what is going on during the prediction process. This lack of interpretability is a huge drawback for using RL in contexts such as healthcare, where understanding and trusting the agent's behavior is as important as its performance.

To overcome this lack of interpretation, interpretable policy representations such as decision trees have gained interest. This is because decision trees are known for their simple, rule-based policies that can be easily understood. After all, the path of any prediction can be traced. However, the problem is that integrating them directly into RL is hard. This is due to standard policy-gradient methods relying on differentiability, while decision trees are non-differentiable because of their hard splits. This means we cannot easily adjust a decision tree using small and smooth changes, which is exactly how most RL algorithms improve their policies. Up until now, strategies such as extracting trees from pre-trained neural network policies like VIPER [6] or relaxing decision trees into differentiable approximations [7, 8] have already been explored. Yet, these methods come with tradeoffs such as reliance on other models, harder interpretability, or a loss of performance during discretization.

Decision Tree Policy Optimization (DTPO), introduced by Vos and Verwer, offers a way to create interpretable policies by combining decision trees with the PPO algorithm [9]. Unlike other methods, DTPO directly builds the decision tree without the need for differentiability. Results in discrete action environments show that DTPO performs competitively with other interpretable models. Also, sometimes it even approaches the performance of deep RL methods, though it may also perform slightly worse depending on the environment.

Still, DTPO has so far only been tested in discrete action spaces, which limits its suitability for many real-world RL problems that involve continuous action spaces. Therefore, this paper aims to answer the following research question:

Can DTPO be extended to support continuous action spaces and remain competitive with neural network policies in terms of performance?

To help guide the answer to the main research question, we formulate three subquestions:

RQ1: How does DTPO perform on discretized continuous action spaces under varying action resolutions?
RQ2: How can DTPO be extended to directly support continuous actions?
RQ3: How does the extended DTPO with continuous actions compare in performance and runtime to RPO with neural networks?

The paper is outlined as follows. In Section 2, the backgrounds on reinforcement learning, deep RL, policy gradient methods, and the DTPO algorithm will be introduced. In Section 3, the methodology of answering the subquestions will be discussed. After that, the results of the corresponding experiments will be analysed in Section 4. Then, in Section 5, the reproducibility of the study and the ethical aspects will be discussed. In Section 6, we provide a discussion of the results, including their implications and limitations. Lastly, in Section 7, this paper will end with the conclusions and the options for future work.

2 Background

2.1 Reinforcement Learning

The difference between RL and supervised learning is that, unlike in supervised learning, the model does not know the correct input-output pairs beforehand and learns through trial and error. Through iterative interaction with the environment, the agent optimizes its cumulative reward by exploring different actions and favoring those that consistently yield higher rewards.

The environment can be modeled as a discounted Markov Decision Process (MDP), which is defined by the tuple $\langle S, A, P, R, \gamma \rangle$ where:

- S is the set of possible states,
- A is the set of possible actions,
- $P(s'|s,a)$ is the probability of moving from state s to s' after action a,
- R_{t+1} is the received reward after action A_t from state S_t,
- $\gamma \in [0,1]$ is the discount factor that determines the importance of future rewards.

So, at each time step t, the agent observes a state S_t, takes an action A_t, and receives a reward R_{t+1}, transitioning to the next state S_{t+1} (see Figure 1).

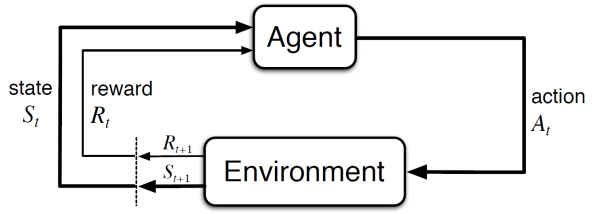


Figure 1: Schematic of the Markov Decision Process (MDP) interaction loop. Adapted from [1].

The behavior of the agent is determined by a policy π, which maps states to a probability distribution over actions. This is denoted as $\pi(a|s)$, and the objective is to find an optimal policy π^* that maximizes the expected return. This return of a policy π is the total accumulated reward from time t:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

To evaluate how good it is to be in a certain state or to take a certain action, we define the state-value function as $v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s]$ and the action-value function as $q_\pi(s,a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a]$. Now, the optimal value functions $v_*(s)$ and $q_*(s,a)$ correspond with the maximum expected return that is achievable from state s, or state-action pair (s,a) respectively.

In this paper, the primary focus will be on policy gradient-based methods. These methods directly parameterize and optimize the policy, which is especially useful when dealing with high-dimensional environments or continuous action spaces.

2.2 Deep RL and Policy Gradient Methods

When we are dealing with environments with small and discrete state and action spaces, it is feasible to store and update values for every possible combination. Methods that use this strategy are called tabular methods. The problem with tabular methods is that they are not scalable if we are dealing with complex environments with high-dimensional state spaces. This is where deep reinforcement learning (deep RL) comes in. Here, deep neural networks are used to approximate the policies or value functions.

A popular deep RL algorithm is the Deep Q-Network (DQN), which approximates the Q-value function $Q(s,a)$ by using a neural network [4]. While this algorithm works well in discrete action spaces, it is less suitable for continuous action spaces where computing a maximum over actions becomes relatively infeasible.

To be able to handle continuous action spaces more effectively, policy gradient methods are often used. In these methods, a parameterized policy $\pi_\theta(a|s)$ is learned where the parameters θ are directly optimized by estimating the gradient of the expected return with respect to the policy. They allow for stochastic policies and support continuous outputs, making it more suitable for RL problems where the number of actions to choose from is high.

From the policy gradient methods, Proximal Policy Optimization (PPO) is one of the algorithms that is widely used due to its combination of performance and stability [5]. By making sure that changes to the policy are clipped during updates, large and unstable jumps are prevented during training. This makes sure that the learning is more consistent overall. In their original paper, PPO showed good performance across different benchmark tasks, including complex continuous control environments in MuJoCo.

However, the main drawback of using (deep) neural networks in RL is their lack of interpretability. Since it is difficult to understand why specific decisions are made, these models are often seen as black boxes. Especially in domains where RL is increasingly used nowadays, like healthcare or autonomous driving, it is important to understand and trust the choices made by the model, as incorrect decisions can have serious consequences.

2.3 Interpretable RL

To address the challenge of interpretability in RL, Bastani et al. [6], Silva et al. [7], and Paleja et al. [8] have looked at more transparent models such as decision trees. In supervised learning, decision trees are popular because they provide simple decisions that humans can easily trace. Starting from the root of the decision tree, a path to a leaf node can be made.

However, integrating decision trees into RL frameworks has its challenges. The main challenge is that most modern RL algorithms, like DQN or PPO, use gradient descent to improve the policy. Similarly to training neural networks, small changes in the parameters are computed based on how much the output (in the case of RL, the reward) changes. Since decision trees contain hard splits, they are non-differentiable, which means we cannot easily compute gradients for their parameters. As a result, standard methods for training deep policies cannot be applied directly to trees.

To tackle this problem, Bastani et al. [6] proposed policy extraction methods, where a decision tree is learned *after* training a neural network policy. An example of this is the VIPER algorithm, which extracts a verifiable decision tree from a pre-trained DQN. A downside of this is that it creates a dependency on the original neural network that is used. Another approach was introduced by Silva et al. [7], and later refined by Paleja et al. [8], who made decision trees differentiable by relaxing the hard decisions into soft decisions. These soft decision trees use functions such as a sigmoid to approximate the splits. While this approach allows for training with gradients, it often makes the trees harder to interpret,

which is why we want to use them in the first place.

2.4 DTPO: Decision Tree Policy Optimization

To overcome the limitations of previously researched methods to interpretable reinforcement learning, Vos and Verwer proposed an algorithm called Decision Tree Policy Optimization (DTPO) [9]. Unlike methods that rely on neural networks or soft decision trees, DTPO directly optimizes a decision tree policy without requiring the model to be differentiable.

The DTPO algorithm is based on the Proximal Policy Optimization (PPO) algorithm [5]. Instead of using gradients to update the parameters of a neural network, DTPO optimizes the structure and parameters of the tree itself. The algorithm treats the decision tree as a policy, where the internal nodes split on state variables and leaf nodes represent actions. It improves its policy by using regression tree learning to update the tree without taking gradients of the parameters. This is similar to gradient boosting, but DTPO updates one tree instead of adding more trees.

Initial experiments have shown that DTPO can achieve competitive performance compared to other interpretable methods such as VIPER, and can even match or exceed the performance of deep RL in some discrete environments. However, it may also perform slightly worse depending on the environment [9]. Thus, the combination of a strong performance and interpretability makes it a good candidate for RL applications.

However, the biggest downside of the DTPO algorithm is that it is only able to handle and has been tested in environments with discrete action spaces. Since many real-world problems involve continuous actions, we are interested in extending DTPO to also handle continuous action spaces and test its performance.

3 Methodology

3.1 Comparison Baseline and Implementation Basis

To evaluate the effectiveness of our continuous-action DTPO variant, we compare it to a strong baseline. Since DTPO is based on the principles of PPO, PPO is a natural starting point for comparison. However, we choose to use Robust Policy Optimization (RPO) instead. RPO is an extension of the PPO algorithm that adds uniform noise to the action mean during training. Aside from this modification, it shares the same architecture and training procedure as PPO. The goal of RPO is to provide a better representation of the action space by encouraging actions with high entropy [10].

In Figure 2, the performance of PPO is compared with the performance of RPO. On the x-axis, we see the total accumulated number of timesteps, and on the y-axis, the mean undiscounted return is shown. The PPO graph is plotted in light blue, and the RPO graph is plotted in dark blue. Here, we see that after around 250 thousand timesteps, the curve for the graph of RPO starts to go up steeper, whereas the curve for the PPO graph stays generally flat. For this reason, we will use RPO as a baseline for performance comparison with the extended DTPO.

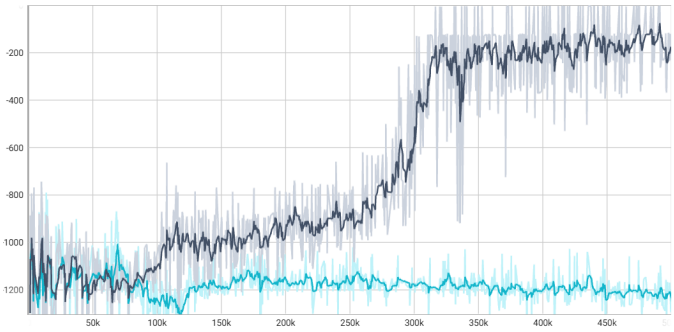


Figure 2: PPO (light blue) vs RPO (dark blue). The cumulative timesteps are plotted on the x-axis, and the mean undiscounted returns are plotted on the y-axis. The learning curve for RPO starts to go up steeper after around 250 thousand timesteps, whereas the curve for the PPO graph stays generally flat.

Although RPO performs better in this environment, we choose to base our continuous-action DTPO implementation on PPO. This is because attempts to integrate action sampling in the same way as RPO into DTPO led to unstable training and poor performance. We hypothesize that this is due to the noise making the training targets inconsistent, since decision trees are much more sensitive to noisy data than neural networks. Therefore, the tree-fitting process becomes unstable, which in turn leads to a degraded policy performance.

3.2 Extending DTPO to Continuous Action Environments

In the paragraphs that follow, we show how we can modify the existing DTPO algorithm[1] to handle continuous action spaces directly, while still keeping DTPO's advantage of interpretability. From now on, we will call this continuous version of the algorithm DTPO-c for a clearer distinction.

First, the original leaf logits in the tree generated by DTPO need to be replaced with a pair of continuous distribution parameters per action dimension. This means that each leaf will need to output a vector of size $2 \times act\_dim$, representing the mean μ and standard deviation σ for each continuous action dimension. To make training easier and deal with fewer constraints, we use the log of the standard deviation since $\log \sigma$ can freely range over $(-\infty, \infty)$, whereas σ itself must be nonnegative. In the end, this value can be exponentiated for better readability when visualizing it in the tree. So, for d action dimensions, a leaf in the tree is interpreted as $[\mu_1, \log \sigma_1, ..., \mu_d, \log \sigma_d]$.

For our implementation, three different configurations (unclipped gradients, clipped gradients, and clipped gradients with a linear decay of the standard deviation) are tested for one seed. Since the combination of clipped gradients (global L2 norm) at 100.0, together with a linear decay of the standard deviation from 1.0 to 0.1 starting from 75 percent of the total iterations until the end of training is the most stable, we will use this variant. The corresponding returns per episode plots can be found in Appendix A

[1] https://github.com/tudelft-cda-lab/DTPO

After that, we need to change how actions are sampled during rollouts. Instead of applying softmax to logits and drawing actions from a categorical distribution, we exponentiate the log-standard deviation to obtain σ and then draw the action from a Gaussian distribution: $a \sim \mathcal{N}(\mu, \sigma^2)$. After drawing the action, we clip it to the environment's bounds.

Although in neural network-based algorithms the reparameterization trick is often used to allow backpropagation through stochastic samples [11], DTPO does not rely on this, since gradient updates are not propagated through the action sampling step. Lastly, for evaluation, we drop the noise term and map each leaf directly to its mean to get a deterministic decision tree.

These changes to the algorithm of replacing logits with $(\mu, \log \sigma)$, sampling actions as $a \sim \mathcal{N}(\mu, \sigma^2)$, and only using the mean for evaluation allow DTPO to work directly with continuous actions. Above all, a single decision tree remains easy to read, and the policy stays interpretable even when dealing with real-valued actions.

3.3 Adapting the DTPO Training Loop for Continuous Actions

To have the tree improve its continuous Gaussian policy, we need to replace the discrete-action PPO loss with a continuous equivalent. Originally, in DTPO, each tree leaf produces logits over which we used a softmax to compute the corresponding action probabilities. Now with DTPO-c, each leaf outputs a mean vector μ and a log standard deviation vector $\log \sigma$ instead, defining a diagonal Gaussian $\mathcal{N}(\mu, \text{diag}(\sigma^2))$. When the agent takes an action $a \in \mathbb{R}^d$, its log-probability under that Gaussian is

$$\ell_{\text{new}}(a; \mu, \sigma) = -\frac{1}{2} \sum_{i=1}^{d} \left[\left(\frac{a_i - \mu_i}{\sigma_i} \right)^2 + 2 \log \sigma_i + \log(2\pi) \right]$$

Subsequently, we compare this probability to the old log-probability $\ell_{old} = \log \pi_{old}(a|s)$, and we get the PPO clipped objective by using

$$L^{\text{CLIP}} = \mathbb{E}[\min(rA, \text{clip}(r, 1-\varepsilon, 1+\varepsilon)A)]$$

where $r = \exp(\ell_{new} - \ell_{old})$ is the probability ratio between the new and the old policies, and A is the advantage estimate.

The reason why we choose to use the log-probability instead of the regular likelihood is that it simplifies mathematical analysis since we no longer need to deal with an exponent that exists in the original Gaussian distribution function. Secondly, because the likelihood function is a product of many small probabilities, it can result in a numerical underflow. However, if we take the logarithm of this likelihood, this product is converted to a sum [12]. Also, since the logarithm is a monotonically increasing function, maximizing the logarithm of a function is equivalent to maximizing the function itself. The full implementation of DTPO-c is available on GitHub[2].

[2]https://github.com/mishakaptein/DTPO-c

4 Results

4.1 Environment Setup and Discretization Strategy

To answer the first research question of how the performance of DTPO is affected by different discretization resolutions, we first need a continuous action environment in which we are going to perform comparisons. For this, we will use the 'Pendulum-v1' environment [13], also known as the inverted pendulum swingup problem. In this environment, one of the ends of a pendulum is attached to a fixed point, and the other end is free. The goal is to apply torque (force) on the free end of the pendulum and swing it into an upright position. Here, the action space is defined as a continuous range between $[-2.0, 2.0]$, and the observation space is an array with three values with their corresponding ranges:

- $x = cos(\theta)$ $([-1.0, 1.0])$
- $y = sin(\theta)$ $([-1.0, 1.0])$
- Angular velocity $([-8.0, 8.0])$

Lastly, the reward is defined as

$$r = -\left(\theta^2 + 0.1 \cdot \dot{\theta}^2 + 0.001 \cdot \tau^2\right)$$

where:

- θ is the angle of the pendulum (measured from the upright position),
- $\dot{\theta}$ is the angular velocity (rate of change of the angle),
- τ is the torque applied by the agent,

This means that the agent is penalized for being far from the upright position, moving too fast, and applying too much torque on the free end.

The next step is to discretize the continuous action space, so that the DTPO algorithm can use it. To analyze the different discretization resolutions, we use a uniform spacing between the actions. The resulting action space for the Pendulum-v1 environment with K actions will then be:

$$Actions = \{-2 + k \cdot \Delta \mid k = 0, 1, 2, ..., K-1\}$$

where the step size is:

$$\Delta = \frac{2 - (-2)}{K - 1} = \frac{4}{K - 1}$$

By using a uniform spacing across the different actions, we make sure that the action space is covered evenly and gives predictable behaviour during training. Several values for K will be evaluated, ranging from low (such as 2 or 3 actions) to high (32 or 64 actions), to see how the performance of DTPO is affected by the action resolution.

4.2 Experimental Setup

To compare the performance of DTPO in a continuous environment, we first modify the DTPO implementation to support varying discrete action sizes. Then, we run the experiments across six different seeds (1, 2, 3, 4, 5, and 6) for every action resolution to get a more accurate estimation of the

means and to lower the standard errors. To ensure the resulting decision tree remains interpretable, we limit the number of leaf nodes to 16.

For DTPO, we use the same hyperparameters as the work of Vos and Verwer: $\eta = 1.0$, $\gamma = 0.99$, $\lambda = 0.95$, $T = 10{,}000$, and $N = 1{,}500$ [9]. All hyperparameters are kept consistent across the different runs for fair comparison, and the undiscounted return is used as the primary evaluation metric. The experiments are executed on a machine with an Intel Core i7-9750H processor and 16 GB of RAM, with each method running on a single core. The full list of all the hyperparameters can be found in Appendix B.1.

4.3 Results of Discretization Study

In Figure 3, the undiscounted return distribution is shown for multiple action resolutions after running the experiment. The full results can be found in Table 1. From both Figure 3 and Table 1, we can see that a finer discretization of the action space does not necessarily lead to better performance.

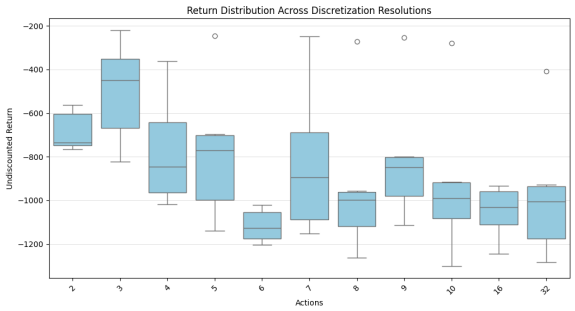


Figure 3: Undiscounted return distribution for various discretization resolutions across six different seeds. A finer discretization of the action space does not necessarily yield better performance.

Table 1: Mean and standard errors of undiscounted returns averaged over six different seeds for a maximum of 16 tree leaf nodes, 10000 simulation steps, and 1500 iterations. Odd-numbered values tend to perform better than nearby even-numbered values on average.

Actions	Pendulum-v1-BangBang
2	-684.43 ± 39.08
3	-500.10 ± 95.11
4	-774.58 ± 104.34
5	-779.59 ± 129.49
6	-1116.45 ± 30.75
7	-826.87 ± 140.26
8	-938.91 ± 141.85
9	-812.78 ± 122.14
10	-927.48 ± 141.71
16	-1051.62 ± 48.65
32	-974.23 ± 126.89
64	-1029.37 ± 119.22

By taking a closer look at the values, one could notice that odd-numbered values tend to perform better than nearby even-numbered values on average. A possible cause of this is that, due to the uniform spacing technique, odd-numbered values contain the 0-action when discretizing the action space. It shows the potential usefulness of applying zero torque when the pendulum is upright and needs to be stabilized. To test this hypothesis, we run the experiment again over the same seeds for the even-numbered values with an additional 0-action *after* discretizing the action space. However, after doing this, we see that it does not have an improved impact on the final return. The results of this second experiment can be found in Table 2.

Table 2: Mean and standard errors of undiscounted returns averaged over six different seeds for even-numbered values plus an added zero action, for a maximum of 16 tree leaf nodes, 10000 simulation steps, and 1500 iterations. Adding a zero action for the discretized action spaces that lack one does not give improvement.

Actions (plus 0-action)	Pendulum-v1-BangBang
4	-885.72 ± 134.51
6	-1022.33 ± 113.61
8	-952.50 ± 85.84
10	-905.43 ± 93.55
16	-1090.20 ± 45.48
32	-1090.60 ± 36.54
64	-1047.21 ± 82.19

4.4 Comparison with RPO

Now that the DTPO-c algorithm is created, we can start comparing its performance with RPO to see if it can reach similar results. Again, the Pendulum-v1 environment will be used to compare the two algorithms, together with three different seeds (1, 2, and 3). For DTPO-c, the maximum number of tree leaves is set to 32 since we assume this to be the highest number of nodes for the tree to still be interpretable. Next to that, we use the hyperparameters $\eta = 1.0$, $\gamma = 0.99$, $\lambda = 0.95$, $T = 10{,}000$, and $N = 2{,}000$ (so a total of 20 million timesteps). For RPO, we use the default parameters as used in the implementation of rpo_continuous_action.py in the CleanRL library[3]. The list of all hyperparameters can be found in Appendix B.2.

4.5 Results of the Continuous DTPO vs. RPO

When comparing the learning curves of DTPO-c and RPO after running the experiments, we can see in Figure 4 that RPO converges to an undiscounted return of about -200 after around 350 thousand timesteps. If we compare this to the undiscounted return per timestep for DTPO-c in Figure 5, we see that it takes significantly more timesteps (12.5 million) to reach the same performance.

But, since DTPO's deterministic tree updates require extra computation per step, it is also worth looking at how the return grows if we look at it from a runtime perspective. In

[3]https://github.com/vwxyzjn/cleanrl

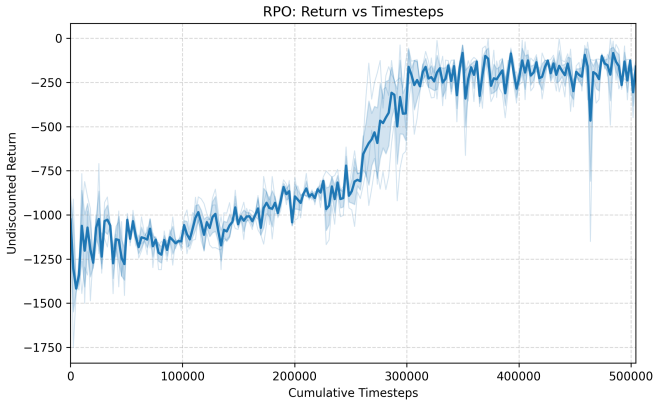


Figure 4: Mean undiscounted return of RPO against the cumulative timesteps averaged across three different seeds. After around 350 thousand timesteps, the return converges to about -200.

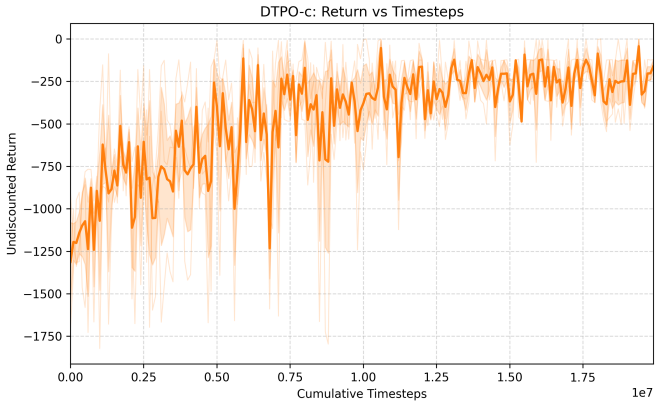


Figure 5: Mean undiscounted return of DTPO-c against the cumulative timesteps averaged across three different seeds. It takes around 12.5 million timesteps to reach the same performance as RPO, which is significantly more than the amount RPO needs.

Figure 6, the mean undiscounted return of both DTPO-c (orange) and RPO (blue) is plotted over the relative time, averaged over three different seeds. We can see that DTPO-c performs better initially, but after approximately 400 seconds, RPO overtakes it again. Overall, Figures 4, 5, and 6 confirm that RPO is more computationally efficient and also has a higher stability during learning in the Pendulum environment than DTPO-c. However, given sufficient runtime, DTPO-c can match RPO in terms of performance. Moreover, DTPO-c produces an interpretable decision tree, as shown in Figure 7, whereas the neural network of RPO remains a black box.

5 Responsible Research

5.1 Ethical Reflection

The focus of this work is on transparency, and since decision trees are fundamentally interpretable, domain experts can inspect each split in the tree to better understand and assess the behaviour of the agent for potential unsafe actions. However,

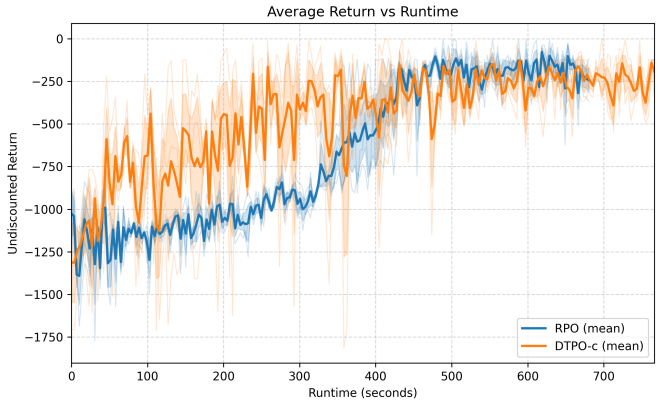


Figure 6: Mean undiscounted return against the relative time averaged across three different seeds. DTPO-c performs better initially, but after approximately 400 seconds, RPO overtakes it again. RPO is more computationally efficient than DTPO-c, but given sufficient runtime, DTPO-c can match RPO in terms of performance.

such a human-readable decision tree can give a false sense of security. This is because unseen states may still lead to unexpected behaviour. Therefore, it is important that such produced decision trees are tested carefully for many different cases.

Another thing to take into account is the potential for adversarial misuse. Individuals who might want to cause harm to the system might identify critical thresholds, after which they can give inputs to the system that exploit edge cases. This means that it might be wise to keep sensitive policy details within a trusted environment.

Finally, deploying interpretable RL in domains that involve human lives (such as healthcare or self-driving cars) raises fairness concerns. In this paper, this is not applicable since our experiments focus on a test control environment. But if a tree (accidentally) splits on an attribute that it is not supposed to, it could introduce bias. Unlike humans, who can consider context and ethical values when making decisions, computers cannot easily do this. Hence, in other environments, it is important that feature choices are verified and that protected information is excluded before training.

5.2 Reproducibility

We have listed all of the code that has been used for this research, together with the corresponding configurations and hyperparameters to perform the experiments. Besides that, we have also listed the GitHub repositories that we used. Therefore, the study is fully reproducible, and the obtained results can be verified. Additionally, the specific random seeds used for the experiments are listed in this paper. Because the experiments are seeded, the same series of random numbers will be generated each time the simulation is run with the same seed.

6 Discussion

This work investigated whether the existing DTPO algorithm can be extended to output continuous-action decision trees to

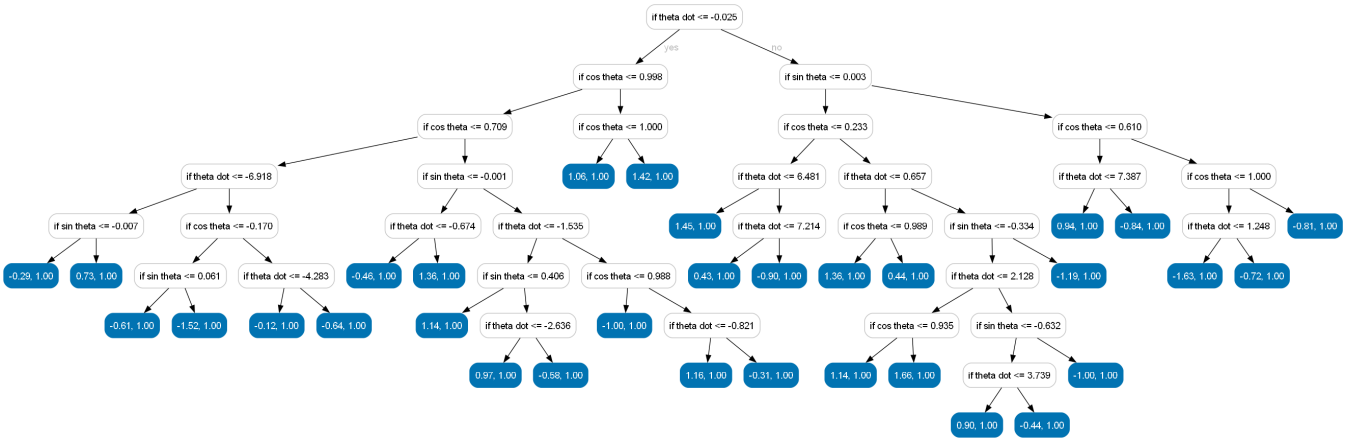


Figure 7: An example of a decision tree outputted by DTPO-c for the Pendulum-v1 environment. The first argument of the leaf nodes represents the torque, and the second argument represents the standard deviation.

handle continuous environments directly, and if this extended variant of the algorithm can match the performance of RPO's neural network policy on the Pendulum-v1 environment.

In our first experiment, we evaluated how varying levels of discretization affect performance using the original DTPO algorithm. The first observation that can be seen from these results is that a finer discretization does not necessarily lead to a better performance, while we might expect that it does, since we are approximating a continuous action space more closely. This likely reflects the trade-off between the number of actions to choose from and model capacity. With a maximum of 16 leaf nodes for the decision tree, it might not be expressive enough to capture optimal policies for large action spaces. Another factor could be that many finely discretized actions are very similar in effect, which makes differentiating between them for the advantage function difficult. In turn, this could lead to a weak or noisy learning signal during policy updates. Lastly, increasing the number of available actions will probably also lead to a higher amount of training data needed to accurately estimate the relative value of each.

After the discretization study, we proposed a modification to the DTPO algorithm (DTPO-c) that can handle continuous action spaces directly and compared it against RPO. The results show that RPO achieves a stable return of around –200 after 350 thousand timesteps, which is approximately 450 seconds. In contrast, DTPO-c initially performs better in early runtime, but eventually RPO overtakes it again after approximately 400 seconds. These results confirm that, although DTPO-c requires more computational effort, it remains capable of matching the performance of RPO with longer training.

Another notable aspect is that the variance of DTPO-c's return is higher than that of RPO. Likely, this is due to the fact that we use a fixed standard deviation of 1.0 in the tree's leaf nodes during the early stages of training, which limits the policy's ability to adapt its exploration. Although a linear decay is used near the end of training, the early fixed value may still lead to less stable learning. Additionally, the discrete structure of the tree may lead to more extreme decision boundaries, making learning less smooth compared to RPO's neural network. Still, DTPO-c offers a fully interpretable policy, which may defend the trade-off in efficiency for applications where transparency and verifiability are crucial.

While these first results may seem promising, there are also some limitations to this study. First, the experiments are only tested on a low-dimensional environment. The results that are formed may not generalize to other higher-dimensional continuous tasks without changing the size of the tree or other configurations. Second, when extending the DTPO algorithm for continuous actions, using a fixed standard deviation in each leaf simplified training but reduced the policy's ability to adapt its exploration across different parts of the state space. Moreover, as noted earlier, DTPO-c required significantly more training steps than RPO to reach a closely similar performance. This highlights a trade-off between interpretability and sample efficiency.

7 Conclusions and Future Work

7.1 Conclusions

In this research, we introduced DTPO-c, a variant of DTPO that can handle continuous action spaces by outputting Gaussian action distributions while preserving interpretability. Through experimentation on the Pendulum-v1 environment, we demonstrated that DTPO-c can achieve comparable performance with RPO, although it requires significantly more timesteps and a longer runtime. This trade-off between transparency and efficiency is key to consider in safety-critical applications.

Our discretization study revealed that a finer discretization of a continuous action space does not necessarily yield better performance, particularly when the tree capacity is limited. Together, these findings support the potential of DTPO-c as a competitive and interpretable solution for continuous control tasks.

7.2 Future Work

There are several promising directions for extending this work. To begin with, the effects of other action space dis-

cretization techniques could be studied concerning the discretization study. In this paper, evenly spaced actions are used. Though, depending on the environment, there exist better alternatives that yield more efficient split decisions in the tree. In addition, it would be worth investigating how the tree capacity, such as the depth or number of leaves, affects the performance across the different discretization levels. This could further clarify the limitations of expressiveness in decision trees.

The DTPO-c variant implemented in this paper can also be tested in higher-dimensional control environments, such as HalfCheetah or Walker. This will help with understanding how the tree depth or leaf count must scale before we lose interpretability. Subsequently, DTPO-c can be modified to allow each leaf to learn its standard deviation, rather than using a fixed or linearly decaying value. This can help with adapting exploration to different parts of the state space and reducing the variance of the return.

Finally, next to improving scalability and expressiveness, future work could also address the limitations in how trees are constructed and used during training. Currently, a new tree is learned from scratch at every update step. This can be inefficient and may cause instability by discarding useful structure from previous trees. Exploring methods that can refine or reuse parts of existing trees could help improve training stability and efficiency.

References

[1] R. S. Sutton and A. G. Barto, *Reinforcement learning - an introduction, 2nd Edition*. MIT Press, 2018.

[2] C. Yu, J. Liu, and S. Nemati, "Reinforcement learning in healthcare: A survey," *CoRR*, vol. abs/1908.08796, 2019.

[3] B. R. Kiran, I. Sobh, V. Talpaert, P. Mannion, A. A. A. Sallab, S. K. Yogamani, and P. Pérez, "Deep reinforcement learning for autonomous driving: A survey," *CoRR*, vol. abs/2002.00444, 2020.

[4] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. A. Riedmiller, A. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nat.*, vol. 518, no. 7540, pp. 529–533, 2015.

[5] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *CoRR*, vol. abs/1707.06347, 2017.

[6] O. Bastani, Y. Pu, and A. Solar-Lezama, "Verifiable reinforcement learning via policy extraction," *CoRR*, vol. abs/1805.08328, 2018.

[7] A. Silva, I. D. J. Rodriguez, T. W. Killian, S. Son, and M. C. Gombolay, "Interpretable reinforcement learning via differentiable decision trees," *CoRR*, vol. abs/1903.09338, 2019.

[8] R. R. Paleja, Y. Niu, A. Silva, C. Ritchie, S. Choi, and M. C. Gombolay, "Learning interpretable, high-performing policies for continuous control problems," *CoRR*, vol. abs/2202.02352, 2022.

[9] D. Vos and S. Verwer, "Optimizing interpretable decision tree policies for reinforcement learning," *CoRR*, vol. abs/2408.11632, 2024.

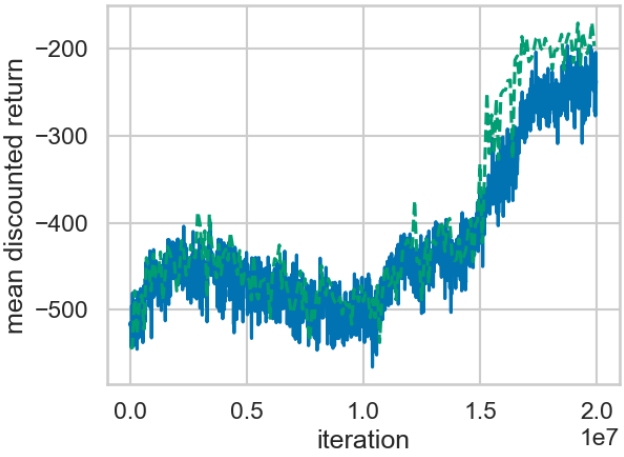
[10] CleanRL, "Robust policy optimization (rpo)." https://docs.cleanrl.dev/rl-algorithms/rpo/, 2025. Accessed: 2025-06-09.

[11] D. P. Kingma and M. Welling, "Auto-encoding variational bayes," in *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings* (Y. Bengio and Y. LeCun, eds.), 2014.

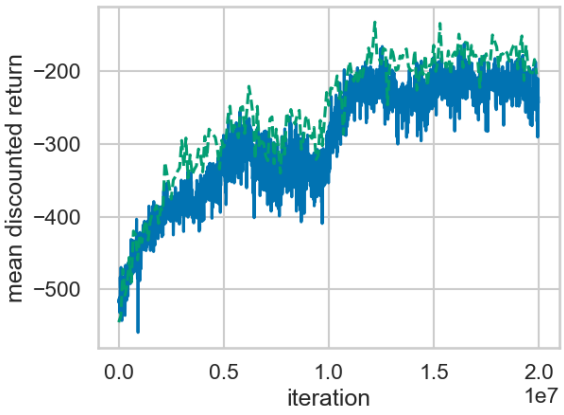
[12] C. M. Bishop, *Pattern recognition and machine learning, 5th Edition*. Information science and statistics, Springer, 2007.

[13] Gym Contributors, "Pendulum - gym documentation," 2022. Accessed: 2025-05-09.

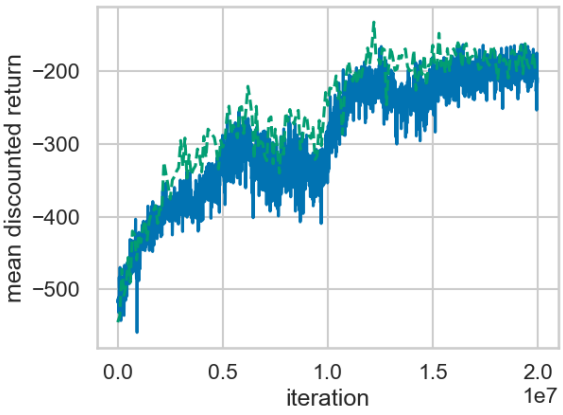
A DTPO-c Configuration Overview



(a) Not clipping the global L2 value and no linear decay of the standard deviation.



(b) Clipping the global L2 value and no linear decay of the standard deviation.



(c) Clipping the global L2 value combined with a linear decay of the standard deviation.

Figure 8: Three different returns per episode plots for various DTPO-c implementation configurations. Clipping the global L2 norm (b) and (c) to a value of 100.0 leads to faster learning compared to not clipping (a). Adding a linear decay for the standard deviation from 1.0 to 0.1 starting from 75 percent of the total training iterations to the end leads to more stability.

B Hyperparameters

B.1 Discretization Experiment

Table 3: Hyperparameters used for DTPO in the discretization experiment.

Hyperparameter	Value
DTPO	
env_name	Pendulum-v1
max_depth	None
max_leaf_nodes	16
simulation_steps	10000
num_envs	1
max_iterations	1500
max_policy_updates	1
ppo_epsilon	0.2
learning_rate	1.0
gamma	0.99
normalize_advantage	True
early_stop_entropy	0.01
evaluation_rollouts	1000
warmup_iterations	0
anneal_lr	False
use_linear_value_function	False
verbose	True

B.2 Continuous Action Comparison

Table 4: Overview of the hyperparameters used for DTPO-c in the comparison.

Hyperparameter	Value
DTPO-c	
env_name	Pendulum-v1
max_depth	None
max_leaf_nodes	32
simulation_steps	10000
num_envs	1
max_iterations	2000
max_policy_updates	1
ppo_epsilon	0.2
learning_rate	1.0
gamma	0.99
normalize_advantage	True
early_stop_entropy	0.01
evaluation_rollouts	1000
warmup_iterations	0
anneal_lr	False
use_linear_value_function	False
verbose	True
grad_clip_norm	100.0

Table 5: Overview of the hyperparameters used for RPO and PPO in the comparison.

Hyperparameter	Value
RPO	
exp_name	rpo_continuous_action
torch_deterministic	True
cuda	False
track	False
wandb_project_name	cleanRL
wandb_entity	None
capture_video	False
env_id	Pendulum-v1
total_timesteps	8000000
learning_rate	0.0003
num_envs	1
num_steps	2048
anneal_lr	True
gamma	0.99
gae_lambda	0.95
num_minibatches	32
update_epochs	10
norm_adv	True
clip_coef	0.2
clip_vloss	True
ent_coef	0.0
vf_coef	0.5
max_grad_norm	0.5
target_kl	None
rpo_alpha	0.5
batch_size	2048
minibatch_size	64
num_iterations	3906
PPO	
exp_name	ppo_continuous_action
torch_deterministic	True
cuda	False
track	False
wandb_project_name	cleanRL
wandb_entity	None
capture_video	False
save_model	False
upload_model	False
env_id	Pendulum-v1
total_timesteps	1000000
learning_rate	0.0003
num_envs	1
num_steps	2048
anneal_lr	True
gamma	0.99
gae_lambda	0.95
num_minibatches	32
update_epochs	10
norm_adv	True
clip_coef	0.2
clip_vloss	True
ent_coef	0.0
vf_coef	0.5
max_grad_norm	0.5
target_kl	None
batch_size	2048
minibatch_size	64
num_iterations	488