Proximal policy optimisation applied to a Mario Kart-like game

AE4350: Bio-inspired Intelligence and Learning for Aerospace Applications

José Cunha

Delft University of Technology

TU Delft

Introduction

The goal of this report is to implement a bio-inspired intelligence algorithm, namely reinforcement learning (RL) using proximal policy optimization. This RL algorithm will be applied to a Mario Kart-like game (made in Unity), where an agent has to drive a kart around a track, with some interactions with the environment. Implementation of the proximal policy optimization algorithm was performed using the Unity Machine Learning Agents Toolkit. The report is divided into three main parts: (i) explanation of the proximal policy optimization algorithm and its parameters; (ii) description of the agents, its environment, and the RL training procedure; (iii) discussion and sensitivity analysis of results. Please refer to the **GitHub repository** to find the code and a description of how to run it.

1 Proximal policy optimisation

1.1. Reinforcement learning

Reinforcement learning (RL) is one of the three main machine learning paradigms, consisting of training agents to take actions by learning from interactions with the environment. Unlike supervised learning (which uses a training set of labelled examples to identify the correct action a system should take) or unsupervised learning (which concerns finding structure hidden in collections of unlabelled data), RL is trying to maximise a reward function by changing and choosing an agent's actions [1].

The general layout of an RL system is where an agent receives rewards (r_t) and a state (s_t) from the environment, and uses those to perform an action (a_t) according to a policy $\pi_\theta(s_t, a_t)$ (mapping from the state to an action) that affects the environment. With each interaction step, the policy is updated, to create actions that maximise the cumulative rewards received. One of the key challenges of RL is the trade-off between exploration and exploitation, wherein to produce a reward, the agent must prefer past actions that produce rewards (exploitation), but to find those actions, it has to explore the potential action space (exploration) which could reduce the current reward [1]. Balancing this trade-off is crucial for effective and stable learning.

Several RL algorithms have been created to find solutions to this exploration-exploitation dilemma, which can be separated into two main categories: value-base and policy-based. Value-based methods focus on estimating the value functions (V^π), and then using that estimate to derive the policy. One notable algorithm in this category is Q-learning [2], which iteratively updates the Q-values (extension of value function with the next action) to find the optimal policy. The other category of policy-based algorithm consists of directly learning the policy mapping states to actions, such as policy gradient methods [3], which promote actions that lead to higher returns (cumulative rewards) and pushes down the probabilities of actions leading to lower returns.

1.2. PPO algorithm

Proximal policy optimisation (PPO) [4] is a RL algorithm which combines value-based and policy-based methods, named an actor-critic approach, wherein the actor updates the policy and the critic updated the value function. PPO improves upon value and policy-based methods by applying more

conservative policy updates, thus improving training stability.

PPO can be characterised by two loss functions Equation 1.1 and Equation 1.4, which denote the actor and critic updates, respectively. The policy objective $\mathcal{L}^{\pi}$ consists of the average minimum of two values: the current-to-old policy probability ratio times the estimated advantage, and the clipped advantage. The clipped advantage can be calculated with Equation 1.2, which clamps the value of the advantage between two values (determined by hyperparameter ϵ), such that policy updates are more conservative. This objective function can then be maximised to provide the policy network parameter updates using gradient ascent methods (e.g. Adam [5]).

$$\mathcal{L}^{\pi} = \mathbb{E}\left\{\min\left(\frac{\pi_{\theta}(a_t, s_t)}{\pi_{\theta,\text{old}}(a_t, s_t)}\hat{A}_t, \quad g(\epsilon, \hat{A}_t)\right)\right\} \tag{1.1}$$

$$g(\epsilon, \hat{A}_t) = \begin{cases}(1+\epsilon)\hat{A}_t, & \hat{A}_t \geq 0\\ (1-\epsilon)\hat{A}_t, & \hat{A}_t \leq 0\end{cases} \tag{1.2}$$

In its policy update, PPO aims to maximise the advantage function (weighted by some value), as it represents which actions are better or worse than average. In general, the advantage can be calculated from the difference between the Q-value and the value function $A(s,a) = Q(s,a) - V(s)$. However, due to the estimate of the value function, this advantage can have a high variance, which adds noise to the policy update. To combat this, an estimate of the advantage function ($\hat{A}_t$) can be used, namely the generalised advantage estimator (GAE). The GAE uses the hyperparameter λ to trade-off between the actual rewards received (with high variance) and the current value estimate (with high bias), and the hyperparameter γ to reduce the contribution of future rewards compared to current rewards. The equation for the GAE can found in Equation 1.3 [6], which uses a cumulative sum of the discounted temporal difference errors (δ_t) for a finite time horizon T.

$$\begin{aligned}\hat{A}_t &= \sum_{l=0}^{T-t}(\gamma\lambda)^l\delta_{t+l} = \delta_t + (\gamma\lambda)\hat{A}_{t+1}\\ \delta_t &= r_t + \gamma V_{\phi}(s_{t+1}) - V_{\phi}(s_t)\end{aligned} \tag{1.3}$$

Moreover, the value function loss can be simply computed from the average square difference between the returns-to-go R_t and the value function $V_{\phi}(s_t)$, seen in Equation 1.4. The returns-to-go represent the actual cumulative reward at a certain point, while the value function is the estimate of the cumulative rewards using parameters ϕ in the critic network. This loss function can then be minimised using gradient descent methods (such as Adam) to provide the value function network parameter updates.

$$\begin{aligned}\mathcal{L}^{V} &= \mathbb{E}\left\{(R_t - V_{\phi}(s_t))^2\right\}\\ R_t &= \sum_{l=0}^{T-t}(\gamma^l)r_{t+l}\end{aligned} \tag{1.4}$$

In sum, PPO is an actor-critic RL method, which consists of two networks (policy and value function) being iterated in parallel to maximise the cumulative rewards, and leading to the optimal policy for the agent. The algorithm can be then summarised in a series of steps[1] to be looped over during training after the two networks (π_{θ} and V_{ϕ}) have been randomly initialised:

1. Collect trajectories using the previous policy $\pi_{\theta,old}$, storing the current state, action, reward and next state.
2. Compute the returns-to-go R_t
3. Compute the advantage estimate $\hat{A}_t$

[1] Example implementation created by OpenAI (spinningup.openai.com) which differs slightly from the Unity Machine Learning Agents Toolkit implementation.

4. Optimise the policy network using the $\mathcal{L}^{\pi}$ objective and gradient ascent
5. Optimise the value function network using the $\mathcal{L}^{V}$ objective and gradient descent

In the following Chapter 2, the Unity Machine Learning Agents Toolkit's [2] implementation of PPO shall be used to train a Mario Kart-like agent to drive around a track. As can be seen from the governing equations of PPO, a series of hyperparameters must be chosen to tune the training for the agent, namely: the discount factor γ, the clipping threshold ϵ, the GAE discount factor λ, the learning rates η for the two network updates and the time horizon T used to sum over the returns-to-go and estimated advantage. The effect of these values shall be considered in Chapter 3.

2
Agent, Environment & Training

2.1. Kart agent and track environment

In order to implement the proximal policy optimisation algorithm, an agent and an environment needs to first be created. As the goal of the assignment is to create a Mario Kart-like agent that can complete a track as quickly as possible, a simplified version of the game must be recreated, in this case, using the Unity[3] platform. The movement of the kart was approximated to that of the original Mario Kart games, and some elements of the track were also implemented, namely off-road sections and track obstacles. All tracks were created using Blender[4] and the free Kenney track assets[5].

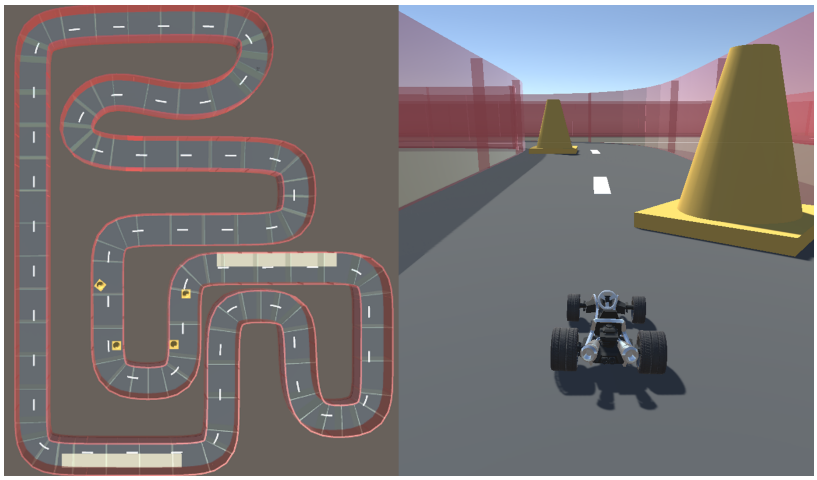

Figure 2.1: *Top view (left) and forward view (right) of the training track environment. Track boundaries/barriers are coloured in red, checkpoints in light green, obstacles in yellow, and off-road in tan.*

To create an environment conducive to train a RL agent, some elements must be added to a track, so that the agent is able to learn in a stable and efficient manner. To make sure that the agent moves in the correct direction, checkpoints are placed throughout the track, which both reward the agent when passing through it, and are detectable by the agent. This greatly accelerates training, as without these, the reward function might have been too sparse, and caused the agent to stay in place to avoid other negative rewards. Another addition are the track boundaries, which restart the training

[2] Unity ML-Agents, github.com/Unity-Technologies/ml-agents
[3] Unity is a cross-platform game engine, unity.com
[4] Blender is an open-source 3D computer graphics software tool, blender.org
[5] A series of free track assets were taken from kenney.nl

episode whenever the agent collides with them. This was done to force the agent to stay on the track, giving negative rewards when the agent does not turn correctly. To add some complexity to the actions of the agent, obstacles can be placed on the track (which must be avoided as they also restart the training episode when hit), as well as off-road sections, which decrease the agent's speed when on them, forcing the agent to avoid these objects. These elements can be seen in Figure 2.1.

Another crucial part of the environment are the rewards, wherein the reward function must be correctly shaped such that the agent is actually able to learn the intended behaviours [7], as the rewards are used to quantify the effectiveness of a certain action [8]. The summary of all the rewards used can be found in Table 2.1, where a reward is only given to the agent when a certain action occurs, except for the last two rewards, which are given each step to promote finishing the lap faster and faster driving - as the kart's maximum speed is 45, it is penalised for speeds lower than 15 (i.e. in off-road sections) and rewarded for faster speeds.

Table 2.1: *Description of reward function and when rewards are applied.*

Reward	Occurrence
0.1	Pass correct checkpoint
1	Finish a lap
5	Finish a lap faster than previous best
-1	Pass incorrect checkpoint
-0.5	Hit track barriers
-1	Hit obstacle on track
-0.1	Reverse kart
-0.002	Given each step
$\frac{\text{speed}-15}{4000}$	Given each step

The main concepts that describe the kart agent are the action space $\mathcal{A}$ and the state/observation space $\mathcal{S}$. $\mathcal{A}$ can be described as a set of six discrete actions, split into two branches (move and turn): move forward, move backward, do not move, turn left, turn right and do not turn. For each iteration of the PPO algorithm, the agent will decide to perform an action for each branch, such that the kart can both move forwards and turn at the same time. Concerning $\mathcal{S}$, there are a total of 95 observations taken from the environment each step, leading to a state space of dimension 95. This consists of the agent's current speed, vector from agent to next checkpoint and the ray-casts sent from the agents that are used to detect the environment. A total of 21 rays (seen in Figure 2.2) are sent each step, where nine are used to detect the barriers and the checkpoints, seven to look down to detect the track conditions (normal or off-road) and five to look far ahead to detect the obstacles and barriers.

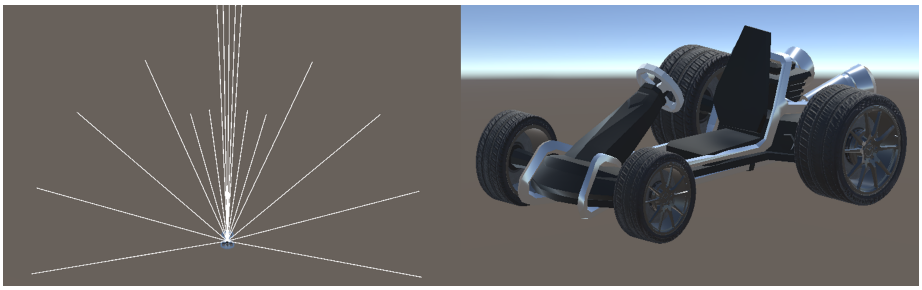

Figure 2.2: *Kart model (right) and visualisation of ray-casts used to detect the environment (left). Kart model taken from Pablo's Lab, youtube.com/c/pabloslab*

2.2. Training procedure

With the agent and the environment defined (namely the state and action space, and the reward function), the training procedure can be constructed, using the aforementioned PPO algorithm to train the agent. With the implementation of the PPO algorithm already provided through the Unity ML-Agents

Python API, one of the most important steps to attain an effective agent is to correctly design the training environment. A well-made training procedure/environment can speed up training (by striking a balance between exploration and exploitation), improve the agent's capability for generalization, and avoid unwanted behaviours in the agent [9].

In the initial part of training, where many of the actions of the agent seem random, it is important that the agent is able to quickly receive some rewards. If not, and solely negative rewards are received, it can happen that the agent will simply do nothing (focus on exploitation) rather than explore the environment for more rewards. For example, the agent will start with a long straight track to learn the 'moving forwards' behaviour, followed by the addition of curves in the track, and then more complex environment characteristics such as off-road sections and obstacles. Several tracks could have been designed to train each of these behaviours subsequently, however the track shown in Figure 2.1 was designed to capture these training steps in a single track.

To further aid in the efficient acquisition of correct behaviours, curriculum learning was applied [10]. This employs the scheduling of the reward signal and of the environment characteristics to encapsulate a certain behaviour the agent should acquire, in order to accelerate learning. Table 2.2 shows the different lessons and when they begin during training.

Table 2.2: *Description of the different lessons in the curriculum learning training procedure, and how each lesson affects the reward signal and the environment characteristics.*

Curriculum	**Rewards**	**Environment**	**Start step**
Lesson 0	Same as Table 2.1, but no speed reward	Only track and barriers	0
Lesson 1	Remove reverse kart reward	Only track and barriers	5×10^5
Lesson 2	Add speed reward	Only track and barriers	2×10^6
Lesson 3	Same as previous	Add off-road and obstacles	5×10^6

Additionally to the curriculum learning, for more efficient training, the PPO algorithm must be carefully tuned. As seen in Chapter 1, there are a series of hyperparameters that alter the behaviour of the PPO algorithm. Possibly the most important concerns the learning rate of the two networks, which scales the size of the policy and value function updates. Both of these were initially set as $\eta = 0.0005$ to encourage faster initial training, and then linearly decreased over time to allow for more stable training as behaviours are fine-tuned. For the PPO clipping threshold, a default value of $\epsilon = 0.2$ was used, as suggested by Schulman et al. [4]. For the reward discount factor, this was set to $\gamma = 0.99$, and is set relatively high as the agent must act in the present in order to 'prepare' for future rewards. The GAE discount factor was also left at its default value of $\lambda = 0.95$, as it led to more stable training. Finally, the time horizon was set as $T = 2000$, as this number of steps is roughly the number of steps required to finish a lap.

Using these algorithm configurations and training procedure, the agents were trained for around 25 million steps, where 20 agents were trained in parallel to speed up training. Further details on the results can be found in Chapter 3.

3
Results & Analysis

3.1. Results

With the training procedure defined, the kart agent can be trained on the track. To best observe the agent's learning, the cumulative (over the episode) reward function can be plotted, which can be seen in Figure 3.1. It should be expected that the cumulative reward increase rapidly at the start, as policy updates are quickly altered from initially random actions. Then, the rewards should start increasing slower, as the policy updates become smaller, in part due to the lower learning rate. As can be seen in Figure 3.1, the cumulative reward does initially increase rapidly, and then tapering off at higher learning steps, however, there is some variability for the higher steps. This can be explained by some training instability caused by the more complex training environment, hinting at some further tuning of the learning rate to improve training stability.

It is also possible to observe how the curriculum learning affects the overall learning. Highlighted in red in Figure 3.1 are the points where the lesson changes, as both after lesson one and two, the rewards quickly increase as the agent begins to receive more positive rewards due to not being punished for moving backwards (which it has mostly already learned not to do) and for maintaining a high speed. Just before lesson three, the cumulative reward seems to be converging, as the agent has been able to fully complete the track, as is slowly improving as it finds the fastest 'racing line' around the track. For lesson three, the cumulative reward initially sharply decreases as the obstacles are introduced, which give a negative reward and end the episode when collided with. Then, the reward keeps increasing, as the agent re-learns the track with the obstacles and off-road present.

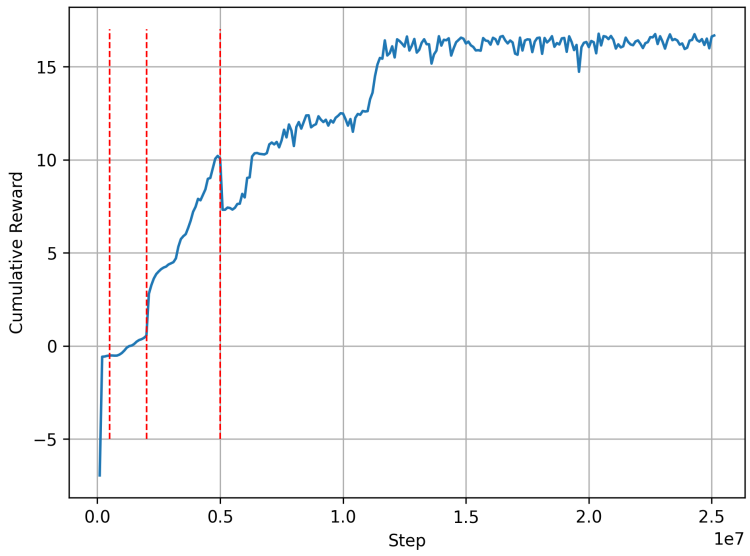

Figure 3.1: *Cumulative reward function of the agent throughout training - highlighted in red are the different lessons.*

Some other interesting results could also be extracted from the training. Considering that each step corresponds to around 0.02 s, the best lap time the agent was able to achieve was 36.6 s (with track obstacles and off-road sections), which surpassed the current human best of 37.4 s. Regarding the accuracy of the resulting model, the agent was able to complete the track 94% of the time, with 98%

of successful runs having a time better than the human best. This relatively high number shows some good generalisability from the agent, as it not only learned the exact track layout or racing line, but was able to actively avoid obstacles, as the position of these are randomized throughout the track, as well as the agent's starting position.

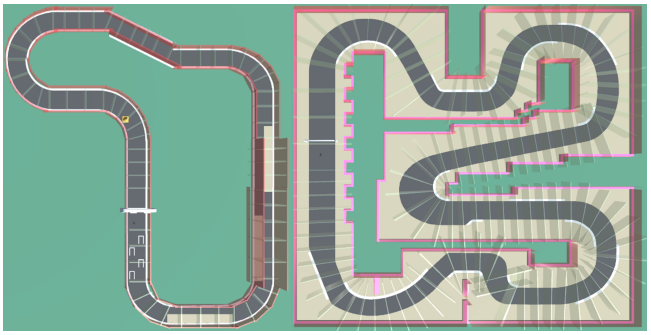

Figure 3.2: *Simpler (left) and more complex (right) validation tracks for the RL model.*

To further test the model's ability to generalise, the same model was placed on a completely unseen track. Two other tracks were created for this purpose: a simpler track with fewer turns and obstacles, and a track with off-road sections around the track and tighter turns, as seen in Figure 3.2. For the simpler track, as all the layout of the track contains elements of the training track, the agent was able to complete it, even without any further training (though at a considerably slower pace than a human driver and with only a reliability of around 14%).

Considering the more complex track, the agent initially is able to slowly go through the corners, though it fails at the 'U-turn' in the middle of the track, as it is a much sharper corner than it has ever encountered. Also, it drives considerably slower than the human driver, as the model has learned to drive close to the track edges (as this provides a time advantage for the training track), which is detrimental in this case, as the track edges are slower off-road sections. This shows that the model in fact does not generalise too favourably, at least without some training for unseen tracks. This result is expected, as found by Packer et al. [11] and Cobbe et al. [12], where agents fail to generalise outside their training environment. To combat this, further environmental stochasticity (for example through procedural generation of the tracks) could have been applied.

3.2. Sensitivity analysis

To observe the effect of the different hyperparameters of the PPO algorithm mentioned in Chapter 1, a sensitivity analysis can be performed. Possibly the most important hyperparameter for any machine learning training procedure is the learning rate (η), as it scales the size of the network parameter updates. Using the initial part of training (before implementing obstacles and off-road) to demonstrate the effect of the learning rate, Figure 3.3a shows the cumulative reward for varying η. For too low learning rates (e.g. $\eta = 5 \times 10^{-5}$), the cumulative reward increases too slowly, where the same level of performance cannot be reached for the same training time. As the learning rate increases, the cumulative reward increases faster, where the best was $\eta = 0.0005$. Increasing the learning rate further begins to cause some training instabilities, leading to slower increases in cumulative reward and a spiking behaviour, with $\eta = 0.005$ not even being able to complete the track.

Considering the changes in the PPO clipping factor, the lower the value, the more conservative the policy update becomes. This causes training to become more stable, though at the cost of slowing down training, which can be seen from Figure 3.3b, as for low values of ϵ, the cumulative reward takes a long time to increase. The recommended value of $\epsilon = 0.2 - 0.3$ seems to yield the best performance, while for higher values, training becomes unstable and the agent never finishes the track. Finally, the reward discount factor, γ, which represents how far into the future the agent should prepare for possible rewards, was varied, which can be seen in Figure 3.3c. For too low discount factors, the agent did not take into account many future rewards, simply trying to maximise rewards

locally, and would therefore constantly collide with the track boundaries. For a γ much closer to 1.0, the agent performed significantly better, as it could prepare for future turns, where the most stable training was achieved with $\gamma = 0.99$.

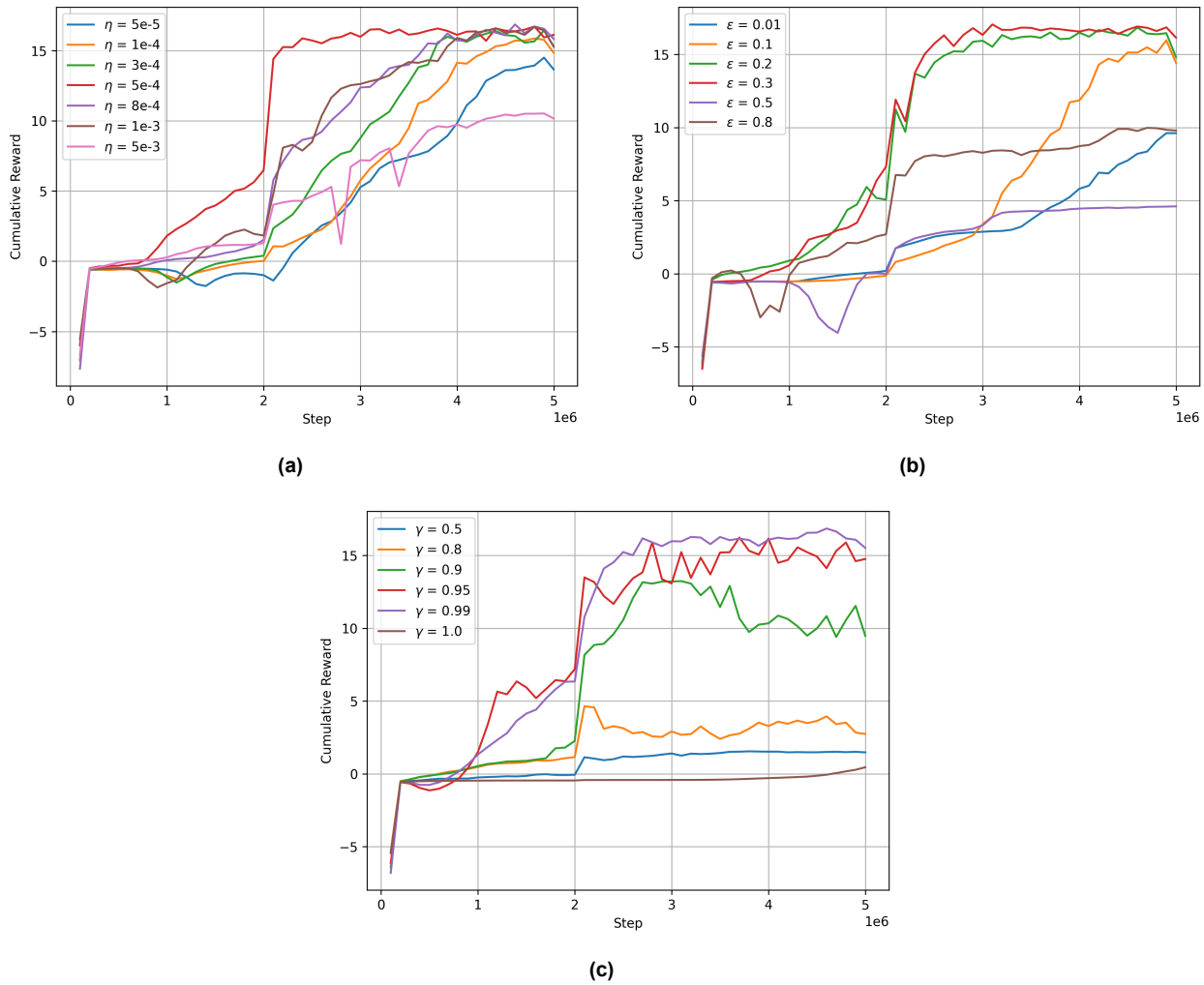

Figure 3.3: *Sensitivity of cumulative reward function to changes in: (a) learning rate, (b) PPO clipping factor, (c) reward discount factor.*

Bibliography

[1] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[2] Volodymyr Mnih et al. “Playing atari with deep reinforcement learning”. In: *arXiv preprint arXiv:1312.5602* (2013).

[3] Richard S Sutton et al. “Policy gradient methods for reinforcement learning with function approximation”. In: *Advances in neural information processing systems* 12 (1999).

[4] John Schulman et al. “Proximal policy optimization algorithms”. In: *arXiv preprint arXiv:1707.06347* (2017).

[5] Diederik P Kingma and Jimmy Ba. “Adam: A method for stochastic optimization”. In: *arXiv preprint arXiv:1412.6980* (2014).

[6] John Schulman et al. “High-dimensional continuous control using generalized advantage estimation”. In: *arXiv preprint arXiv:1506.02438* (2015).

[7] Ahmed Abouelazm, Jonas Michel, and J Marius Zoellner. “A Review of Reward Functions for Reinforcement Learning in the context of Autonomous Driving”. In: *arXiv preprint arXiv:2405.01440* (2024).

[8] Benjamin Evans, Herman A Engelbrecht, and Hendrik W Jordaan. “Reward signal design for autonomous racing”. In: *2021 20th International Conference on Advanced Robotics (ICAR)*. IEEE. 2021, pp. 455–460.

[9] Siyue Zheng. “The influence of different environments on reinforcement learning”. In: *2022 3rd International Conference on Computer Vision, Image and Deep Learning & International Conference on Computer Engineering and Applications (CVIDL & ICCEA)*. IEEE. 2022, pp. 401–406.

[10] Sanmit Narvekar et al. “Curriculum learning for reinforcement learning domains: A framework and survey”. In: *Journal of Machine Learning Research* 21.181 (2020), pp. 1–50.

[11] Charles Packer et al. “Assessing generalization in deep reinforcement learning”. In: *arXiv preprint arXiv:1810.12282* (2018).

[12] Karl Cobbe et al. “Quantifying generalization in reinforcement learning”. In: *International conference on machine learning*. PMLR. 2019, pp. 1282–1289.