Teaching the Go1 Robot to Walk
Reinforcement Learning Project

Stefan Lechner, Dario Spoljaric, Zoltán Varga, Benedikt Frey
{e01608096, e11806417, e11823287, e12230445}@student.tuwien.ac.at

***Abstract*—This paper explores the application of reinforcement learning (RL) to teach the Unitree Go1 robot to walk in the Multi-Joint Dynamics with Contact (MuJoCo) framework. The study focuses on comparing different neural network architectures, including Multilayer Perceptron (MLP) and Long Short-Term Memory (LSTM), as well as evaluating the impact of Proximal Policy Optimization (PPO) with Prioritized Experience Replay (PER) and varying reward functions. The results show that a PPO implementation with a PER replay buffer and an MLP neural network with four layers achieves optimal convergence. The study contributes to our understanding of effective RL approaches for robotic locomotion.**

***Index Terms*—Reinforcement Learning, Proximal Policy Optimization, Prioritized Experience Replay, Long Short-Term Memory, Locomotion, Robot Learning**

I. Introduction

In the pursuit of stabilising complex systems, e.g. robotic dogs, classical control theories reach their limits quickly. To precisely tune a PID or LQG controller for each joint to walk, a thorough understanding of the system is necessary to ensure smooth interaction between components. A different way of solving this problem is to simulate every possible system interaction with its environment and observe what works best. This procedure is called reinforcement learning (RL). It requires a significant amount of computational power, nonetheless, it is becoming an increasingly popular solution thanks to recent advances in affordable and powerful computer components. The key advantage of RL is that it is often easier to model the interaction between the system and the environment than to model the entire system on its own, allowing for more efficient optimisation of the system's behaviour. However, this approach also presents its own set of challenges, such as the risk of getting stuck in local optima, the need for a large amount of data to train the model, and the difficulty in ensuring safety and robustness in the learned behaviour, especially in real-world applications with complex and dynamic environments.

RL is a type of machine learning that enables an agent to learn optimal policies through trial-and-error interactions with an environment [1]. Figure 1 illustrates the basic idea behind the environment feedback loop.

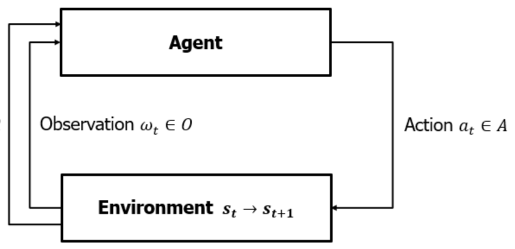

Fig. 1. Reinforcement Learning Feedback Loop

An agent selects an action a_t based on its current state s_t and receives positive or negative rewards r_t from the environment as feedback. The observation ω_t the agent receives captures the state change that occurs when performing an action. The agent's objective is to learn a policy that maximises the cumulative reward over time. This is achieved by iteratively updating the policy based on the feedback received from the environment, allowing the agent to improve its performance and make better decisions over time. The goal is to find an optimal policy that balances the trade-off between exploration and exploitation.

II. Related Work

There are two main types of RL algorithms. A value-based and a policy-based strategy [2]. The former calculates so-called q values, which define a value for a chosen action a in a state s. To maximise the rewards over time, the q function gets calculated at every time step. Through iterative execution, the agent implicitly learns the optimal policy $\pi(s)$ for each state. The policy-based method does not directly calculate an optimal action but follows a predefined policy $\pi_\theta(a_t \mid s_t)$ and a policy objective function $J(\theta)$. The objective function is defined as the sum over all time steps t of all expected, discounted rewards if the strategy $\pi_\theta(a_t \mid s_t)$ is followed. The aim is to maximise the rewards. This is done by adjusting the policy through gradient descent. Both methods have some advantages and some problems.

The solution for more optimal performance is a compromise between value-based and strategy-based approaches. This method is known as the actor-critic method. Here, the "Actor" selects an action according to a certain strategy. This strategy is defined by a policy-based method. The "Critic" is implemented as a value-based method that evaluates the action of the actor. To put this into equations, an advantage function is defined.

$$A(s,a) = Q(s,a) - V(s), \tag{1}$$

This function states how much better or worse the reward for a particular action is than the average reward for an action in this state. A new gradient of the goal function can then be defined (2).

$$g = \mathbb{E}_t \left[\nabla_\theta \log \pi_\theta(a_t \mid s_t) A(s_t, a_t) \right], \tag{2}$$

The value function is weighted higher if there is a large positive or negative advantage and weighted lower if a smaller advantage has been calculated. The Critic therefore "evaluates" the action of the actor. The actor-critic approach combines the advantages of both methods. High sampling efficiency and low variance, similar to the value-based approach. In addition, the method can also be applied in time-continuous environments in the same way as in policy-based approaches.

A. Proximal Policy Optimization

To begin with, it is important to acknowledge that optimizing multiple steps simultaneously can lead to significant strategy updates. While this approach can be effective in certain situations, it also has its drawbacks. Specifically, if several optimization steps are carried out at once, it can result in major strategy updates that can have a destructive effect on the learning process. To address this issue, the Proximal Policy Optimization (PPO) approach has been proposed [3]. To understand PPO, it is important to first examine the Trust Region Policy Optimization (TRPO [4]) approach. As the name implies, a "trust region" is calculated, which limits the size of the policy updates depending on how big the trust region is. This approach achieves very good performance but is complex to implement and requires many estimators or high computing power. To simplify this approach and achieve similar performance, the PPO approach is introduced. First, the likelihood ratio between the current strategy and the previous strategy is defined (3).

$$r(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}, \tag{3}$$

This ratio replaces the old score function (2), while the advantage function is retained. Finally, the new method should also limit strategy updates. To this end, the ratio is clipped as soon as it is outside a defined interval. This removes the incentive to change the strategy too much. The new function is also known as the surrogate objective function.

$$L^{\text{CLIP}}(\theta) = \mathbb{E}_t \left[\min\left(r(\theta) \cdot A_t, \text{clip}(r(\theta), 1-\epsilon, 1+\epsilon) \cdot A_t \right) \right], \tag{4}$$

In the surrogate objective function, the smaller of the two terms is chosen to estimate the expected value. This means that negative policy changes are still taken into account while positive changes outside the interval are cut off.

Proximal Policy Optimization is one of the most widely used approaches in the field of reinforcement learning [5], as the approach delivers comparable performance to TRPO but is much easier to implement [3].

B. Prioritized Experience Replay

Schaul et. al [6] introduced Prioritised Experience Replay (PER) to optimise the use of saved experiences in the replay memory. In temporal-difference (TD) problems, which include RL problems, experiences are saved and used to improve the strategy. The saved experiences are sampled uniformly for the learning process. In PER, as the name suggests, important experiences get prioritised. The equation (5) describes with which probability the i-th experience is sampled for the replay storage. Where α specifies how much prioritisation is used.

$$P_{\text{sample}}(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha} \tag{5}$$

The priority p_i of the i-th experience is calculated from the absolute temporal-difference error, which means cases with a high impact, positive or negative impact the priority most. In the case of a PPO implementation, TD error is calculated as the advantage function, so the priority of an experience is defined $p_i = |A_i| + \epsilon$, with ϵ a small factor to prevent edge-cases, where $p_i = 0$. This prioritised sampling introduces bias, which uncontrollably influences the stochastic distribution. To correct this problem weights are introduced (6).

$$w_i = \left(\frac{1}{N} \cdot \frac{1}{P_{\text{sample}}(i)} \right)^\beta \tag{6}$$

Where N is the total number of experiences in the replay buffer.

The prioritised sampling is fully compensated for $\beta = 1$. Schaul et. al "hypothesise that a small bias can be ignored in this context" [6] and suggest *β-annealing*. A process in which β increases over time and only reaches 1 at the end of the learning process. The calculated weights can be directly used in the Q-Learning process by weighting the TD-error $w_i A_i$ while updating the loss function.

The authors show that the PER approach outperforms a traditional Deep-Q-Network for various problems.

C. Long Short-Term Memory

If we consider nature's way of quadrupedal movement as the golden standard, one can immediately notice a periodic pattern in the movement of humans and animals. To make use of this time dependency and translate it to neural networks, it might be useful to have some sort of embedded memory block. There are various types of Recurrent Neural Networks of which we have chosen the Long Short-Term Memory model (LSTM) due to its simplicity.

The LSTM [7] provides a simple and computationally efficient way of dealing with time-variance. It retains the last couple of steps as long as the model determines it to be relevant. In each recurrence, it re-considers which information to keep and which to forget. The newly calculated state of the cell is saved for the next iteration and reset at the beginning of a new run.

LSTMs have been known to perform well in some reinforcement learning tasks [8] including motion planning [9] as well. In this particular task, there is no long-term time dependency, only a short periodic pattern. We expect the model to recognise this periodicity and help the agent to achieve an animal-like movement quicker than traditional dense neural networks.

III. Method

This chapter briefly describes the method used in this project. The environment in which the robot interacts is explained, as well as the agents and the reward structure. Defining all elements of the reinforcement learning structure is shown in figure 1.

A. Environment

The Multi-Joint Dynamics with Contact (MuJoCo) framework [10] is a general-purpose physics engine that is used, for example in the field of robotics, biomechanics and machine learning. MuJoCo is a highly optimised simulation environment that offers numerous solver parameters. This allows users to adjust and adapt to any model [11]. The model can be created using a MuJoCo native XML format. This project uses a URDF model of the Unitree Go1 robot [12] that is loaded into a bare environment with just a floor to walk on. The MuJoCo engine gives the user access to the joint position and joint velocity, as well as other robot hyperparameters, such as stiffness or damping. The action space for this project is defined as the vector of all joint positions, meaning the actor can set the rotation angle for all 12 joints of the Go1 robot. The observation space – additionally to the joint positions – includes joint velocity as well. In order to speed up the learning process we limited the action space of the abduction, hip and knee to be in the ranges specified in table II (in the appendix). In MuJoCo there are different types of actuators, we used position-driven actuators which use PD controllers to calculate the torque applied to the motors. The controller parameters have been set in the XML file for abduction, hip and knee independently and according to the previous work [13].

B. Agent

The agent is implemented as an Actor-Critic model. The PPO approach and the addition of the PER buffer are already explained in the Related Work section. Additionally, the advantages of the PPO implementation were calculated using the Generalized Advantage Estimation (GAE) method [14]. GAE is used to reduce variance in calculating the advantage estimation. Also, as found in other implementations [15] gradient clipping to limit the policy updates and Kullback-Leibler (KL) divergence to dynamically adjust the learning rate were implemented. KL divergence "quantifies the proximity of two probability distributions" [16]. This idea can be used to increase the learning rate for a higher divergence and reduce it if the divergence between the old policy and the new policy is small.

During the course of this project, different implementations for the actor-critic agent were tested. The results of the different approaches were documented and are presented in the next chapter.

C. Reward

Reward engineering is an important and very time-consuming part of the reinforcement learning process. The agent inherently learns to maximise rewards by optimising the loss function. In our study, we adopt reward functions similar to those employed in previous works, such as [17], [18], [19], [20], and [21]. The baseline rewards used in our project are outlined in Table I. Specifically, we set the command for angular velocity ω_{cmd_z} to 0.0 and for linear velocity v_{cmd_x} to 0.5.

TABLE I
Baseline rewards

Reward	Equation r_i	Weight w_i
Lin. velocity tracking	$e^{-4(v_{cmd_x}-v_x)^2}$	30.0
Ang. velocity tracking	$e^{-4(\omega_{cmd_z}-\omega_{yaw})^2}$	5.0
Linear velocity (z)	$v_z 2$	-2.0
Orientation	$\theta^2+\psi^2+\phi^2$	-50.0
Feet up reward	$(p_{z,k}-p_{fz,k})^2 v_{f_x y,k}$	-10
Action rate	$(a_t-a_{t-1})^2$	-0.01
On track reward	$x \cdot e^{-\lvert\lvert\mathbf{y}-\mathbf{y_{des}}\rvert\rvert}$	10.0
Termination penalty	$\begin{cases} 1, 0.5 < z < 0.2 \\ 1, \lvert\phi\rvert/\lvert\psi\rvert/\lvert\theta\rvert > \frac{\pi}{3} \\ 1, \lvert y\rvert > 0.3 \\ 1, x < -0.3 \end{cases}$	-20

The total reward is then computed by

$$r_t = \sum_{i=1}^{n} r_i \cdot w_i \tag{7}$$

These rewards I are common in the locomotion terrain [17] [18] and will serve as the foundation for analysis of our learning algorithms in the following sections. To mitigate velocity or coordinate drift from a desired but static target, squared error metrics are employed, as we did in the z-direction velocity and orientation rewards. Exponential functions, as can be seen in the tracking rewards, are employed to follow a given command, which is sampled throughout the training process. In that way the robot not only learns to follow one specific target but an arbitrary one. In our case, this command is just a simple forward velocity. We even designed several reward functions ourselves, one that proved to accelerate learning was the "on track reward" which leads the robot to maximise its x-component while maintaining a small y-component. More reward functions have been tested and the results will be shown in the next chapter.

IV. Evaluation

A. Actor-Critic Implementation Comparisons

In this section, we will dive into the differences in the total reward and loss metric by comparing the baseline code, the PPO and the PPO with PER implementation. The baseline implementation uses a standard replay buffer and calculates rewards in a straightforward manner. It was made available to us as a starting ground. Specifically, it calculates the temporal difference by combining the current rewards at a given step with the product of a boolean condition, the discount factor, and the subsequent values. It then assigns this delta to the advantages array at the specified step and computes the returns by adding the advantages and the current values. The other methods were already explained in the beginner sections of this paper.

We begin with analysing our best implementation of PPO + PER against the baseline implementation. Since the baseline does not produce any meaningful loss, the comparison is based on the maximised total reward, as shown in Figure 2. For each run, we trained the robot for 700 epochs. While the reward of our implementation is rising consistently the reward of the baseline is collapsing and does not improve in the slightest. This is also confirmed by the visualisation of the robot where PPO + PER learned a walking motion and the baseline implementation did not.

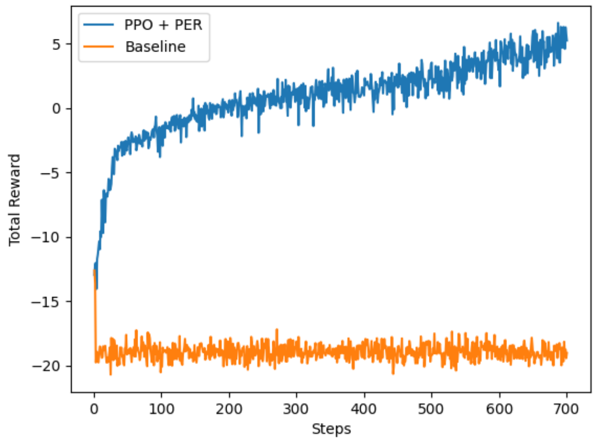

Fig. 2. PPO/PER against Baseline Implementation

Second, we need to understand if the PPO implementation on its own would have been enough to teach a robotic dog a walking motion. For this, we changed the alpha value of the PER implementation where 0 is purely random and 1 leads to strict selection of what is stored in the replay buffer. Instances of 0, 0.6, and 1 as alpha values are depicted in Figure 3 showing the loss of our actor and value networks. The idea behind a loss plot is that the loss should converge to zero because it indicates how well the network reduces the gradient. After analysing the blue lines from the PPO-only implementation, we noticed that the value loss remains stagnant, and the actor loss, which initially moves towards zero, starts increasing again. This indicates that there is a problem with the learning process. On the other hand the actor and value loss from the 0.6 and 1 alpha implementation converge towards zero. One exception is the actor loss of PER with alpha set to 1 which rises slightly again from epoch 350 on, underscoring that our best implementation is with alpha as 0.6. Interestingly all implementations lead to a walking robotic dog. There is only a difference if one looks at each reward individually. The PPO/PER implementation learns them faster than the implementation without the PER code. This was expected and we can say with confidence that the PER implementation chooses instances of high learning possibilities.

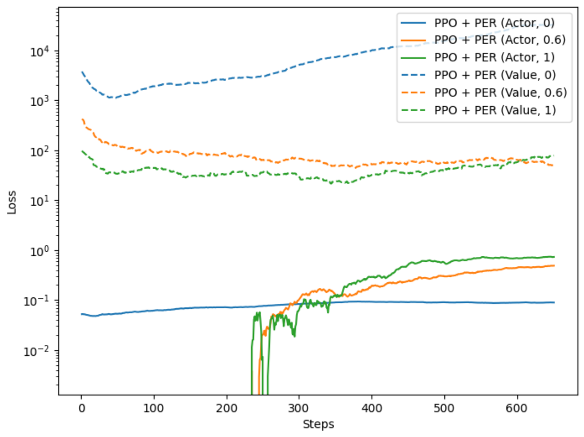

Fig. 3. PPO/PER change of alpha value (randomness)

B. Comparing Neural Network Implementations

To find the best network for the specified problem, we have tested two different architectures with varying depth. The first approach was a Multilayer Perceptron consisting of 3 to 5 layers, the second one was an Long Short-Term Memory module with a dense projection layer at the end and two before the memory cell to prepare the feature vectors. Both models had a hidden dimension size of 512.

After training the model for at least 700 epochs, it was tested in the MuJoCo environment to see how many epochs it needs to learn progressing forward and to follow the path given in the reward functions. The training was performed using an Adam optimiser with a learning rate of 0.001. Further details about the RL agent is described in III.

The comparison is based on three metrics. Figure 4 shows the number of epochs needed to achieve a desired movement. The first time the robot has left the starting position was noted in each training phase. To make the comparison more accurate, a second metric was introduced. This denotes the first epoch where the robot has progressed forward without falling or rotating to the side for at least a couple blocks in the simulation environment. The third comparison metric is an evaluation of the environment's mean total reward of the last ten epochs (690–700). For detailed results see Figure 5.

Out of the different MLP settings, the one with 4 layers came out as the best model according to two of the three metrics. This was then compared to the LSTM as actor and as critic in separate tests. We have found out that the LSTM does not work as actor — the robot could not move at all, even after 700 epochs. MLP as critic could make the robot walk, although it performed slightly worse than the approaches using MLP only. This contradicts our expectations regarding RNNs in Reinforcement Learning. The reason behind it may be that LSTM's memory range was too long and it therefore could not improve the RL agent's ability to achieve a smooth and periodic movement. Another reason might be that the environment's rewards were tuned using an MLP and then switched to LSTM without defining other reward functions that could be more compatible with an RNN-based critic.

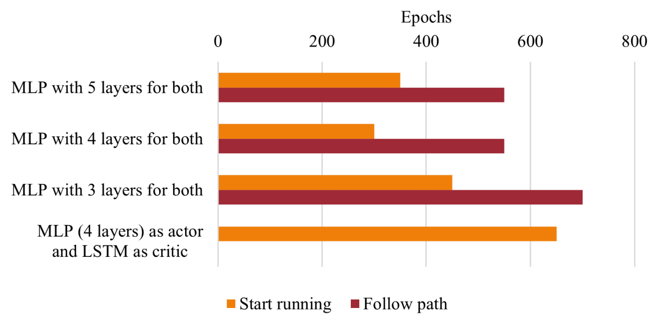

Fig. 4. Comparison of different neural networks as actor and as critic

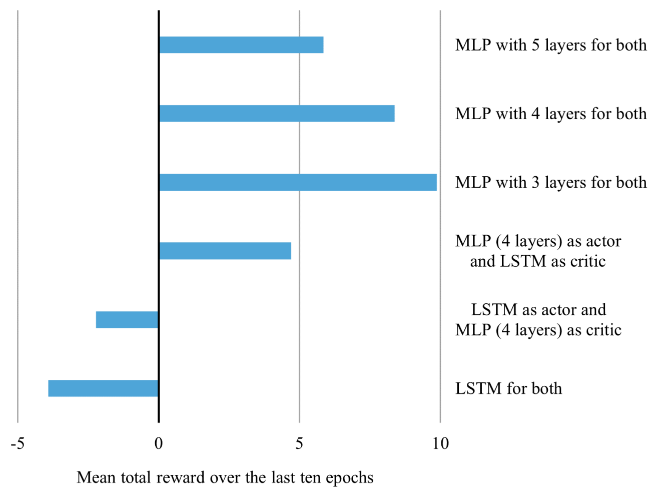

Fig. 5. Comparison of different neural networks based on the achieved reward

The loss functions for the MLP and LSTM implementation look very similar 11, 12 (in the appendix). LSTM seems to converge slightly stricter towards zero, but the difference is small.

C. Evaluating the reward functions

For this section, the findings of previous sections will be used. These suggest that an MLP with four layers in total, coupled with PER incorporating the setting of $\alpha = 0.6$, exhibits optimal convergence for both value and actor loss. The goal of this project is a walking robot that is able to hold a straight line. To show the effectiveness of our optimised rewards, we compare our baseline I and our modified version of the MuJoCo tutorial [22] against DreamWaQ [17]. Adjustments were made to the weighting of MuJoCo tutorial rewards and one slight change to the DreamWaQ rewards, as outlined in Tables IV and III. However, the weighting of DreamWaQ overall was kept the same. The comparison of total rewards, illustrated in Figure 6, reveals that both the default implementation and the MuJoCo tutorial implementation outperform DreamWaQ.

The most interesting reward in our regard is the "linear velocity tracking" reward, as it is most representative of the success of the model. A closer examination of the reward composition (Figures 8, 9, and 10 in the appendix) reveals that the reward consistently increases for "default" and "mujoco" throughout the training process, making them outperform "DreamWaQ". While one might argue that this discrepancy is attributed to the reward weighting – with "default" and "mujoco" implementations placing more emphasis on linear velocity and termination, for example – the results speak for themselves.

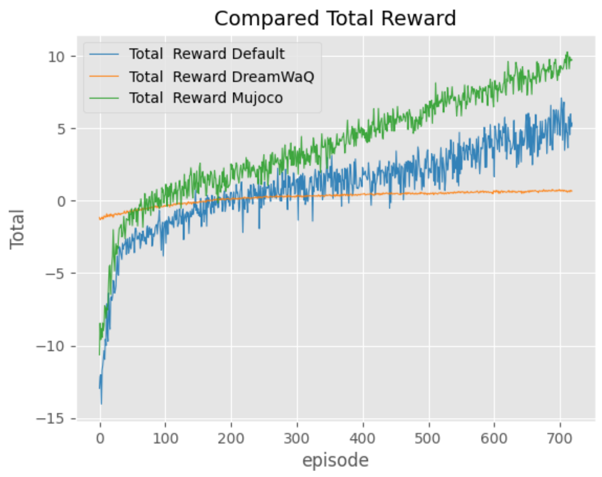

Fig. 6. Comparison of rewards

Notably, after 700 epochs, the Go1 robot demonstrates successful walking with both the default and MuJoCo implementations. In contrast, the DreamWaQ implementation fails to achieve walking even after 1000 epochs. This underscores the importance of weighting and shows the superior performance of our and MuJoCo's implementation.

D. Batch fitting

In the final step, we experimented with the batch size for the replay buffer. The number of experiences in the replay buffer (batch size) is determined by the number of time steps per epoch that were used and the predefined number of batches.

$$batch_{size} = \frac{N_{timesteps}}{N_{batches}} \quad (8)$$

Figure 13 (appendix) shows the rewards gained for different batch sizes. This suggests that as the number of time steps and batch size increase, the accumulated rewards also increase. But in our experience this correlation is limited.

In Figure 7, we depict the cumulative rewards over a consistent number of time steps ($N_{timesteps} = 1000$). Above all, the highest cumulative rewards are achieved with the largest batch size (500) for a batch count of 2. However, it is essential to highlight that the performance drops significantly for the next largest batch size (5 batches), performing much worse compared to smaller batch sizes (achived with 10 and 20 batches).

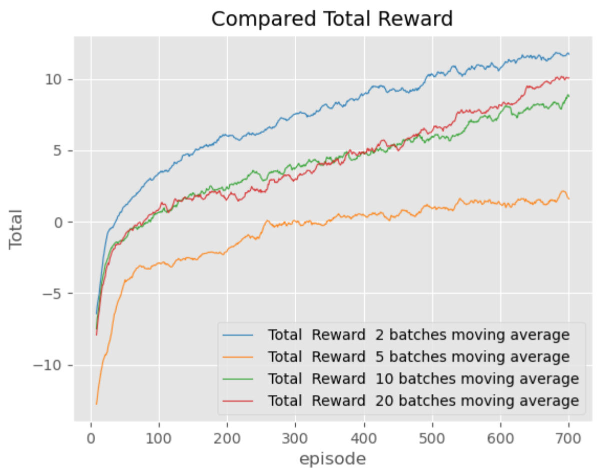

Fig. 7. Rewards for varying number of batches

This indicates that there is not a direct correlation for batch size to performance but rather a sweet spot. In our case this spot was found to be at size 500 which relates to a time of 20 seconds in simulation.

To yield optimal training in the future the batch size will be varied to find the optimal solution for the problem.

V. Discussion

In this paper, we demonstrated the process of teaching the Unitree Go1 robot a walking motion in a MuJoCo simulation environment. The robot learned a straightforward motion after less than 600 episodes for a PPO implementation with a PER replay buffer and 4 total layers (2 hidden layers). We also looked at an LSTM method for the neural network implementation of the Actor-Critic implementation. Interestingly, the LSTM implementation led to slightly better convergence of the loss function, which did not translate to the learning process or the accumulated rewards and varying results for the same implementation. This also coincides with the findings of researchers, suggesting that deep RL methods are unreliable across runs, hard to reproduce and can be brittle [23]. We also examined different reward functions. The MuJoCo implementation augmented with our own reward functions led to the smoothest walking motion in the simulation. The walking motion is already massively improved in comparison to previous implementations. The total rewards 6 showed further learning progress even after 600 epochs meanwhile DreamWaQ stagnated.

In future work, we would like to improve the walking motion. The goal is a more ”animal-like walk” or a jumping motion. There are already different implementations that show promising results. We could achieve that by randomisation of walking commands as it was done in [22] so the network learns to follow these instead of just one behaviour. Another exciting aspect in terms of reward engineering are potential-based rewards [24] which mitigate a more generalizable and natural walking behaviour to the network. Further parallelised learning as mentioned in [20] and [25] could further stabilise the learning process and lead to a much better explored environment. We also worked on the environment to create a more realistic one with stairs (14) or other obstacles from STL files (15). The goal is to have a robot that can manage different terrain and navigate a planned path.

References

[1] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, "A brief survey of deep reinforcement learning," *arXiv preprint arXiv:1708.05866*, 2017.

[2] D. Silver, "Lectures on reinforcement learning," URL: https://www.davidsilver.uk/teaching/, 2015.

[3] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," 2017.

[4] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel, "Trust region policy optimization," 2017.

[5] Y. Wang, H. He, and X. Tan, "Truly proximal policy optimization," in *Uncertainty in Artificial Intelligence*. PMLR, 2020, pp. 113–122.

[6] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," 2016.

[7] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, pp. 1735–1780, 1997.

[8] B. Bakker, "Reinforcement learning with long short-term memory," in *Advances in Neural Information Processing Systems*, T. Dietterich, S. Becker, and Z. Ghahramani, Eds., vol. 14. MIT Press, 2001. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2001/file/a38b16173474ba8b1a95bcbc30d3b8a5-Paper.pdf

[9] M. Everett, Y. F. Chen, and J. P. How, "Motion planning among dynamic, decision-making agents with deep reinforcement learning," in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2018, pp. 3052–3059.

[10] E. Todorov, T. Erez, and Y. Tassa, "Mujoco: A physics engine for model-based control," IEEE, pp. 5026–5033, 2012.

[11] M. Körber, J. Lange, S. Rediske, S. Steinmann, and R. Glück, "Comparing popular simulation environments in the scope of robotics and reinforcement learning," 2021.

[12] UniTree. (2024) Go1. Accessed: 18.01.2024. [Online]. Available: https://m.unitree.com/go1/

[13] G. Feng, H. Zhang, Z. Li, X. B. Peng, B. Basireddy, L. Yue, Z. Song, L. Yang, Y. Liu, K. Sreenath, and S. Levine, "Genloco: Generalized locomotion controllers for quadrupedal robots," 2022.

[14] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, "High-dimensional continuous control using generalized advantage estimation," 2018.

[15] G. B. Margolis and P. Agrawal, "Walk these ways: Tuning robot control for generalization with multiplicity of behavior," *Conference on Robot Learning*, 2022.

[16] J. Shlens, "Notes on kullback-leibler divergence and likelihood," 2014.

[17] I. M. A. Nahrendra, B. Yu, and H. Myung, "Dreamwaq: Learning robust quadrupedal locomotion with implicit terrain imagination via deep reinforcement learning," 2023.

[18] S. H. Jeon, S. Heim, C. Khazoom, and S. Kim, "Benchmarking potential based rewards for learning humanoid locomotion," in *2023 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, May 2023. [Online]. Available: http://dx.doi.org/10.1109/ICRA48891.2023.10160885

[19] Y. Kim, H. Oh, J. Lee, J. Choi, G. Ji, M. Jung, D. Youm, and J. Hwangbo, "Not only rewards but also constraints: Applications on legged robot locomotion," 2024.

[20] N. Rudin, D. Hoeller, P. Reist, and M. Hutter, "Learning to walk in minutes using massively parallel deep reinforcement learning," 2022.

[21] J. Lee, J. Hwangbo, L. Wellhausen, V. Koltun, and M. Hutter, "Learning quadrupedal locomotion over challenging terrain," *Science Robotics*, vol. 5, no. 47, Oct. 2020. [Online]. Available: http://dx.doi.org/10.1126/scirobotics.abc5986

[22] G. Deepmind, "Mujoco tutorial," 2023, accessed on 21.01.2024. [Online]. Available: https://colab.research.google.com/github/google-deepmind/mujoco/blob/main/mjx/tutorial.ipynb

[23] L. Engstrom, A. Ilyas, S. Santurkar, D. Tsipras, F. Janoos, L. Rudolph, and A. Madry, "Implementation matters in deep policy gradients: A case study on ppo and trpo," *arXiv preprint arXiv:2005.12729*, 2020.

[24] S. H. Jeon, S. Heim, C. Khazoom, and S. Kim, "Benchmarking potential based rewards for learning humanoid locomotion," in *2023 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, May 2023. [Online]. Available: http://dx.doi.org/10.1109/ICRA48891.2023.10160885

[25] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," 2016.

VI. Appendix

TABLE II
JOINT LIMITS OF ABDUCTION, HIP, AND KNEE

Joint	Range of Motion
Hip Abduction	[-0.3, 0.3]
Hip Extension	[-0.686,1.8]
Knee	[-2.0,-0.8]

TABLE III
DREAMWAQ WITH ADJUSTED WEIGHTS

Reward	Equation (r_i)	Weight (w_i)
Lin. velocity tracking	$e^{-4(v_{cmd}-V_{ay})}$	1.5
Ang. velocity tracking	$e^{-4(w_{ond}-w_{yaw})}$	0.5
Linear velocity (z)	$v_{z_o}^2$	-2.0
Angular velocity (xy)	ω_{xu}^z	-0.05
Orientation	$-\dot{\theta}^2$	-0.2
Joint accelerations	-2.5×10^{-7}	$-2.5e-7$
Joint power	$\|\tau\| \left\|\dot{\theta}\right\|$	$-2.5e-5$
Body height	$(h_{des}-h)^2$	-1.0
Foot clearance	$(pez_k - Pf_{z,k})^2 V_{f,xy,k}$	-10.0
Action rate	$(a_t - a_{t-1})^2$	-0.01
Smoothness	$(a_t - 2a_{t-1} + a_{t-2})^2$	-0.01
Power distribution	$var(T_0)^2$	$-1e-7$

TABLE IV
MUJOCO TUTORIAL INSPIRED REWARDS

Reward	Equation (r_i)	Weight (w_i)
Lin. velocity tracking	$e^{-4(v_{cmd}-V_{ay})}$	30
Ang. velocity tracking	$e^{-4(w_{ond}-w_{yaw})}$	5.0
Action rate	$(a_t - a_{t-1})^2$	-0.01
Termination penalty	$\begin{cases} 1, 0.5 < z < 0.2 \\ 1, \|\phi\|/\|\psi\|/\|\theta\| > \frac{\pi}{3} \\ 1, \|y\| > 0.3 \\ 1, x < -0.3 \end{cases}$	-20
Linear velocity (z)	$v_{z_o}^2$	-2.0
Feet air time	$\sum t_{feet\ in\ air}$	0.2
Torque reward	$\sqrt{\sum_{i=1}^N \tau_i^2} + \sum_{i=1}^N \|\tau_i\|$	$-2 \cdot 10^{-4}$
Stand still	$(q_t - q_{init})^2, x_{cmd} < 0.1\frac{m}{s}$	-0.5
Angular velocity (xy)	$\omega_x^2 + \omega_y^2$	-0.05
Foot slip	$\sum^{feet\ higher}(p_{offset} + v_i)^2$	-0.1
Orientation	$\theta^2 + \psi^2 + \phi^2$	-50.0
On track reward	$x \cdot e^{-\|\|\mathbf{y}-\mathbf{y_{des}}\|\|}$	10.0

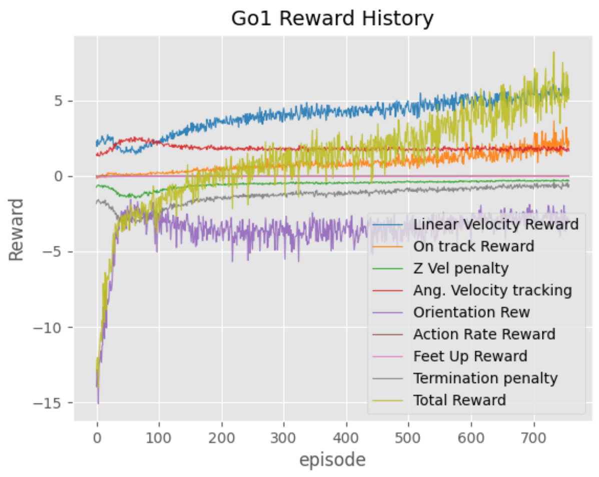

Fig. 8. Default Rewards

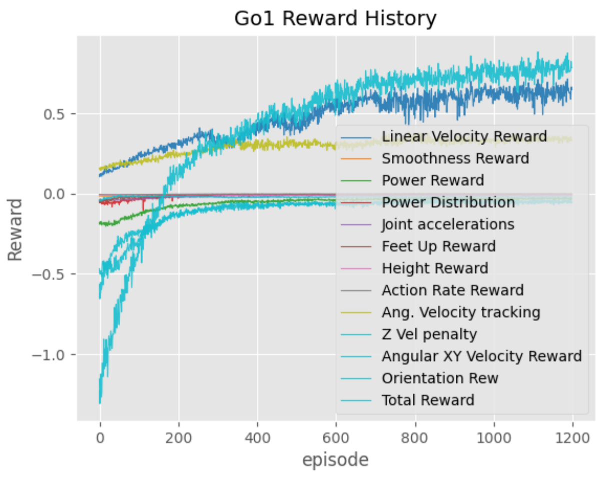

Fig. 10. DreamWaQ Rewards

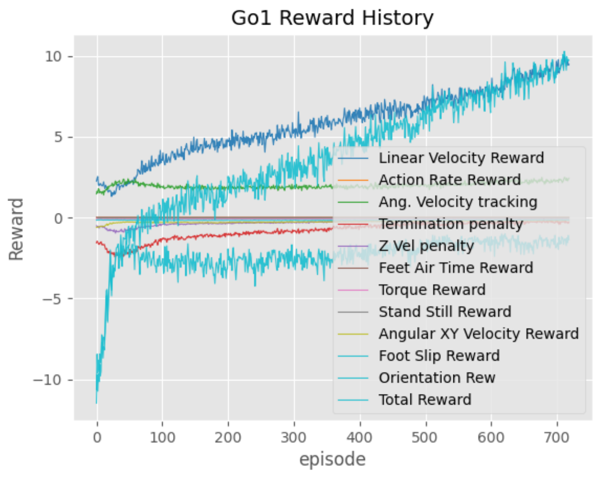

Fig. 9. Mujoco Rewards

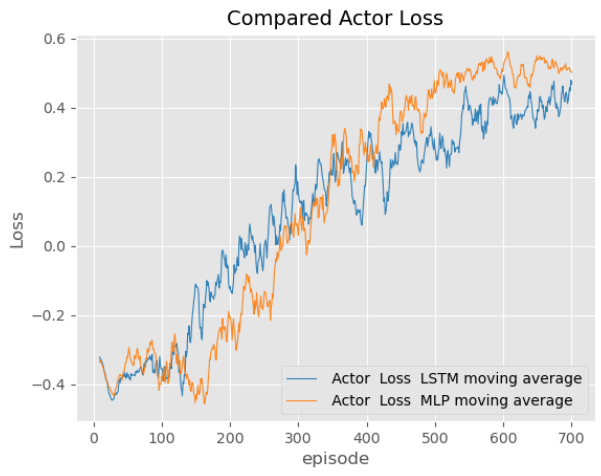

Fig. 11. Actor loss comparison LSTM and MLP

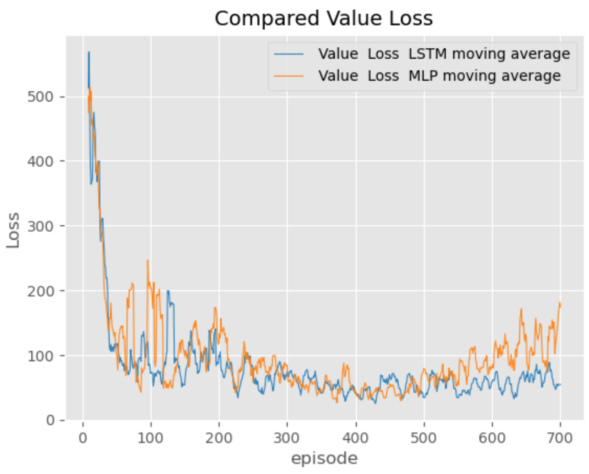

Fig. 12. Value loss comparison LSTM and MLP

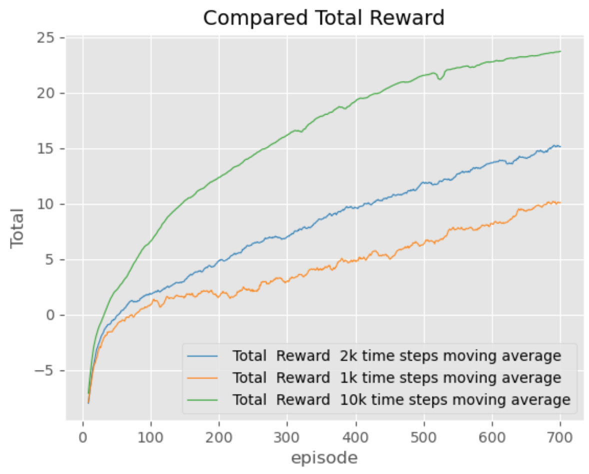

Fig. 13. Compared rewards - 20 batches rising number of time steps

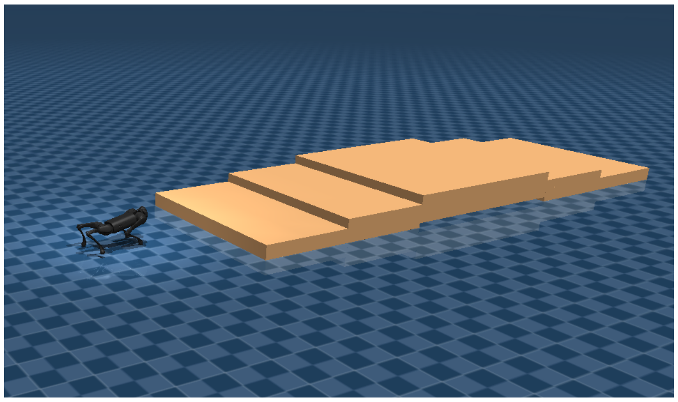

Fig. 14. Mujoco environment with stairs

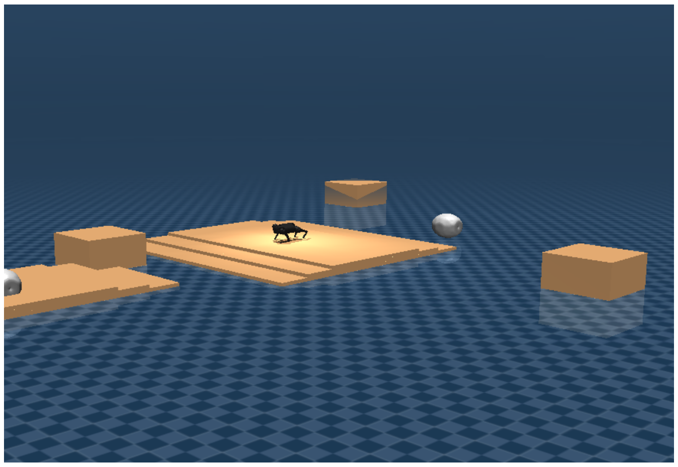

Fig. 15. Mujoco environment with stl objects