Project the robot

Gustav Pråmell, Samuel Wallander Leyonberg, Jacob Danielsson, Matilda Ronder

October 21, 2025

1 Abstract

For a robotic system to behave in a dynamic environment, deterministic algorithms may not be sufficient or feasible as it is not possible to state each state-action pair for complex tasks. Deep reinforcement learning percents solution's for this, as it learns to predict its next state given the current state and action taken. For robotic systems, this means that complex tasks such as balancing and walking on four legs can be achieved. However, not all reinforcement learning (RL) algorithms are created equally. Depending on the problem at hand, different types of RL need to be implemented. For four legged robotic systems, often called quadrupled robots or robot dogs, there is a need for a RL-algorithm that works with continuous state and action spaces in order to solve complex tasks. This paper covers the final methodology and result from the underlying work where, we (the authors) explores how to enable a walking behavior in a simulated quadruped robot using deep reinforcement learning. This paper describes the problem space, how the problem is presented in a physics-based simulated environment, description of the RL-algorithms being used as well as there implementation and lastly, the final result.

2 Introduction

Quadruped robots are robotic systems with four primary means of movement, typically in form of legs. These systems are often referred to as *robot dogs* due to there structural and functional resemblance. They often implement mobile robotic systems developed to replicate the movement pattern of four-legged animals[8]. Their compact size and reduced weight make them suitable to integrate into environments where humans may otherwise

be harmed[3]. Furthermore, they are useful in environments such as complex terrain, where wheeled or tracked robots often struggle.

Training the behavior control of such robot is often done by using reinforcement learning (RL) algorithm. By implementing a quadruped robot within a simulated physic-based environment, behaviors such as walking and balancing can be learned through interaction with the environment.

The purpose of this study is to enable a quadruped robot to learn and execute walking behavior within a simulated physics environment towards a set goal within the environment. The goal are defined as coordinates placed a few meters from the initial robot position and shown as a red box or ball for visitational purposes. The simulated environment are created with Pybullet, a module for physics simulation where articulated bodies from URDF and other file formats can be loaded[1]. Within this environment, a model of the AlienGo quadrupled robot[7] was implemented as the physical body to interact with the simulation environment. A screenshot of the PyBullet simulated environment containing the AlienGO quadruped robot and the red *goal box* can be seen in Figure 1.

Other studies[10, 9, 4] that have developed a quadruped robot with reinforcement learning have shown promising results in learning control policies from interaction with the environment. Among these, Proximal Policy Optimization (PPO) has been a popular algorithm due to its balance between training stability and computational efficiency.
In this project, two different reinforcement learning algorithms were applied to train the robot. The Proximal Policy Optimization (PPO) and Soft Action-Critic (SAC) algorithm. These algorithms showed improvements in different areas. PPO demonstrated positive training progress in addressing the balance and standing problem, while SAC performed better in terms of reaching the goal. The analysis of these algorithms provides insights into designing efficient training strategies for quadruped robots.

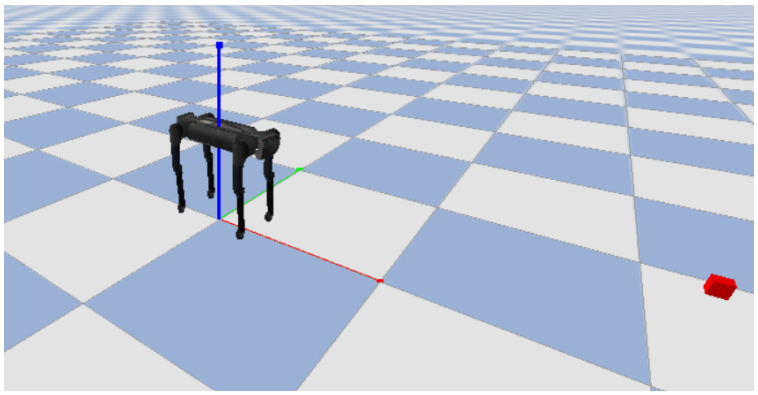

Figure 1: AlienGO quadruped robot in PyBulet physics engine.

3 Methodology

3.1 Problem Description

The reinforcement learning problem is formulated to enable a quadruped robot to learn stable and efficient walking behavior within a simulated environment. The agent, representing the robot, interacts with a physics-based simulation with the end goal of reaching predefined coordinates.

Quadrupedal locomotion is often formulated as a reinforcement learning problem, in the framework of Markov Decision Processes (MDP). MDP is defined as the tuple (S, A, R, P, γ), where S represents the set of observable states, A is the set of possible actions, R is the reward function, P is the joint probability of a reward *r* and the next state *s'* (given a state and action) and γ is a discount factor. The state *s* is observed by the robot at each time step *t* and an action *a* is derived from the robots policy π.

However, in practice, the robot does not have access to the full underlying state of the environment. Instead, the state space includes continuous variables such as joint angles, joint velocities, body orientation and distance to the goal. Though a lot more can be observed, the magnitude of possible observations that can be implemented makes it not feasible to oversevere them all. Therefore, the agent is in a continuous partly observed environment.

3.2 Environment

The training environment is an open area with a flat ground plane set in PyBullet. The robot was placed in the center of the environment a few centimeters above ground, so its first contact with the ground became a part in the training process. This setup allowed the agent to learn both balance and recovery during landing, to later on learn moving towards a set goal. A minimalistic environment was chosen to narrow the walking task difficulties, ensuring that the robot could focus on learning stable locomotion without additional challenges from other obstacles or bumpy terrain.

The robot consists of four legs and a body. Every leg has three joints, one at the knee, one at the hip and the third one at the ”shoulder”. This makes it possible to move the legs in a ”3D space” such as x, y, z positions. The legs are controlled by the joint torque, which control the speed and movement at each joint. The action space defines what the robot can control or change at each time step. This is linked to the joints and their torques or positions. The torque value consists of 12 values, 3 joints per leg x 4 legs. The agent

output these torques, which through a PID-controller, changes the legs positioning and consequently the walking behavior. The reward function is used to give the robot feedback after each action. It tells the robot how good or bad the action was and depending on that, it receives positive or negative reward. Over time, it should learn how to maximize the total reward, meaning it discovers actions that lead to the goal and successful walking. The joint probability represents the likelihood of two or more random variables occurring together, often states, actions, rewards and next states. In this case, the agent is in state *s*, takes action *a*, receives reward *r* and transitions to state *s'*. The discount factor tells the agent how much it should value future rewards compared to immediate ones and it goes between 0 and 1.

3.3 Proximal Policy Optimization (PPO)

PPO is an on-policy algorithm that can handle continuous action and state spaces. This makes it suitable for training locomotion in quadruped robots, where the robot operates in a continuous environment and each joint receives torque commands. In addition to being on-policy, PPO uses an actor-critic architecture, which provides more stable updates when the critic is trained correctly. It also employs bootstrapping, which reduces variance and improves sample efficiency. Pseudocode[5] for PPO could be found in subsection *Pseudocode PPO*.

To enhance the training of the PPO the usage of parallel environments were used, 16 environments ran parallel to each other with an individual robot in each. The environment were normalized since this helps stabilize training and can prevent features from dominating the gradient updates made by the PPO, this makes the networks landscape smoother and more predicable. The rewards of the enviroment were also normalized to control the magnitude of the rewards. This because of the scale dependence that PPO:s clipping mechanism have which if the rewards are unstable very small or very large this can lead to unstable updates.

Pseudocode PPO

Algorithm 1 PPO-Clip

1: **Input:** initial policy parameters θ_0, initial value function parameters ϕ_0
2: **for** $k = 0, 1, 2, \ldots$ **do**
3: Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
4: Compute rewards-to-go $\hat{R}_t$.
5: Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.
6: Update the policy by maximizing the PPO-Clip objective:

$$\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min \left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t), g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \right),$$

typically via stochastic gradient ascent with Adam.
7: Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2,$$

typically via some gradient descent algorithm.
8: **end for**

Before updating the policy, a set trajectories needs to be collected as seen in step 3. These trajectories do not need to cover the agents full episode, but can be batched over a predefined number of time steps. This batching allows multiple updates during one episode. Walking can be seen as a repeating set of actions, batching can make for faster learning if implemented correctly.

3.4 Soft Action-Critic (SAC)

A SAC (Soft Actor-Critic) environment was set up in the following way, see Pseudocode[6] under subsection *Pseudocode SAC*. Following the parallel environment option previously, a vector with 48 environments was created. In addition to being able to save progress during runtime, and to be able to monitor how the training is going. An evaluation environment was created that would run the current model on a set time step interval to see how it would score in a solo environment. In addition, a callback function for saving the model in a set

time step interval was also created. With this setup the model was going to be doing the following. Model training for *t* time steps while saving and evaluating on set intervals during training. This enables the model to be monitored and stopped if needed by us, since the runtime was several hours long.

During the training of the SAC model, different reward functions were tested, but the final model that achieved the best results used the following rewards:

- *Forward reward. Rewarding the robot for having a positive speed forward.*
- *Progress reward. Moving forward with positive speed and moving towards the goal.*
- *Upright reward. Rewards the robot for staying upright.*

Other rewards that have been tested but not used in the best versions are pitch reward, roll reward, stable reward, height reward, and distance to goal reward. All these aim to either provide the robot with a reward for standing on all four feet and being stable or give it rewards for moving towards the goal.

The currently best model is trained from scratch, where the dog would spawn a small distance above the floor and then have to learn from that. Other models have also tried using a base model to start the training from where it would already be able to stand or move around small distances, and from that point start the training process.

For hyperparameter for the SB3 SAC model there has not been a lot that changed from default. Learning rate, batch size, tau and Replay buffers was looked at during the course of the project. While most of these could add more stability to the model it would also increase the training time and for this project that was not possible. But to gain some stability, learning rate was slightly altered even though it was probably came at a time cost.

Pseudocode SAC

Algorithm 2 Soft Actor-Critic (SAC)

Input: initial policy parameters θ, Q-function parameters ϕ_1, ϕ_2, empty replay buffer $\mathcal{D}$
Set target parameters equal to main parameters $\phi_{\text{targ},1} \leftarrow \phi_1$, $\phi_{\text{targ},2} \leftarrow \phi_2$
repeat
    Observe state s and select action $a \sim \pi_\theta(\cdot|s)$
    Execute a in the environment
    Observe next state s', reward r, and done signal d to indicate whether s' is terminal
    Store (s, a, r, s', d) in replay buffer $\mathcal{D}$
    If s' is terminal, reset environment state
    if it's time to update **then**
        for j in range (however many updates) **do**
            Randomly sample a batch of transitions, $B = \{(s, a, r, s', d)\}$ from $\mathcal{D}$
            Compute targets for the Q functions:

$$y(r, s', d) = r + \gamma(1 - d)\left(\min_{i=1,2} Q_{\phi_{\text{targ},i}}(s', \tilde{a}') - \alpha \log \pi_\theta(\tilde{a}'|s')\right), \quad \tilde{a}' \sim \pi_\theta(\cdot|s')$$

            Update Q-functions by one step of gradient descent using

$$\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(Q_{\phi_i}(s, a) - y(r, s', d)\right)^2, \quad \text{for } i = 1, 2$$

            Update policy by one step of gradient ascent using

$$\nabla_\theta \frac{1}{|B|} \sum_{s \in B} \left(\min_{i=1,2} Q_{\phi_i}(s, \tilde{a}_\theta(s)) - \alpha \log \pi_\theta(\tilde{a}_\theta(s)|s)\right),$$

            where $\tilde{a}_\theta(s)$ is a sample from $\pi_\theta(\cdot|s)$ which is differentiable w.r.t. θ via the reparameterization trick.
            Update target networks with

$$\phi_{\text{targ},i} \leftarrow \rho \phi_{\text{targ},i} + (1 - \rho)\phi_i, \quad \text{for } i = 1, 2$$

        end for
    end if
until convergence

4 Results

4.1 Proximal Policy Optimization (PPO)

The environment was designed so that the robot was initially placed a few centimeters above the ground. This setup allowed the robot to first learn to catch itself during the fall

and subsequently to maintain balance after landing. This was done with the taught of phase training where the final goal of the agent being able to walk towards a target would proceed from a standing phase to a walking phase and continued to a walking towards a target. The PPO configuration that performed best with this standing phase Regarding balance and standing, this model performed best. The model was trained on the following hyperparameters:

- **Learning rate:** $3 * 10^{-4}$
- **Batch size:** 64
- **Total training time steps:** 100 000
- **Number of parallel environments:** 16 (DummyVecEnv)
- **Reward function:** $0.5 * forward\_reward + 1.0 * height + 2.0 * up\_vector + 0.8 * forward\_vel - (5 * distance\_to\_target) - 10 * backward\_penalty$

Other parameters that are not mentioned here are initialized with the default values that the PPO comes with from the stablebaselines3 package[2].

4.2 Soft Action-Critic (SAC)

Based on the setup discussed in previous sections, a model was trained on 50 million time steps. This model was able to ”seal walk” towards the set goal. Meaning it sat on its back legs and walked with the front legs to gain ground. Even if this is not the optimal solution, it is a clear sign that it is doing something right, and with more time, it would be able to fine-tune the reward function even more to achieve greater results. The model was trained on a time step limit of 4000 that cut the robots training session early before able to reach the goal. In this model, that was a problem because it nearly reached the goal but was killed before reaching it. This can be seen in the demo, since when it reach the end of the part it has been trained on it just falls over.

The best model were trained on these hyperparameters and reward function:

- $+0.6 * r\_forward$
- $+1.2 * r\_progress$
- $+0.5 * r\_upright$
- $-0.001 * np.sum(np.square(action))$

- **Learning rate:** 10^{-4}
- **Batch size:** 256
- **Replay buffer size:** 10^6
- **Soft update rate (τ):** 0.005
- **Total training time steps:** 50 million
- **Number of parallel environments:** 48 (DummyVecEnv)

5 Discussion

5.1 PPO

For the problem that faced the model of making the robot being able to stand up and balance showed to be a harder problem for the model than anticipated. The model was able to interact with the joints of the robot in a couple of ways torque control which controls the amount of torque to turn the joint into desired position, position control which sets the position of the joint or velocity control which is the speed of the joint. The model were restrained with a couple of conditions that would terminate the episode. These were pitch, roll which terminated the episode if model controlled the robot into a angle which make him lean or tilt which is not a suitable stance for the robot with the goal of standing still/balancing. Height were a termination criteria since a low height would show that the robot has fallen and vice versa with a high height that the robot has elevated. Time steps were also an ending criteria to have an ending episode for not having an possibility of an infinite run.

These termination conditions were also used for the reward shaping of the model. The intended rewards for standing up were constructed from belief of what the standing or balancing state required from the robot. Such as an upright position compared to the ground with a penalty for leaning or tilting to ensure an upright standing position. A combination reward were introduced which combined a low tilt, pitch and yaw together with a height constraint for keeping the robot in a desired height to enforce a standing and upright position of the robot. This were one of the few rewards the model could get to try to guide the model towards a standing position. This was done in comparison with more detailed reward function for a specific standing stance, a forward tilt, number of contact points from the legs and many more, this resulted in more of the model exploiting specific rewards for

finding a maxima in those. This often resulted in a stance where the agent did a sort of a split and balanced low to the ground or outright just fell, which did not result in a standing pose of the robot.

The solution for enabling the model to stand and balance was to use the PPO algorithm while narrowing down the reward function to limit the robots ability to exploit undesired rewards.

Since these parameters showed promising progress early in training, logging methods were used for the PPO runs. The training duration was short enough that the results could be manually evaluated using the returned logs.

5.2 Soft Action-Critic (SAC)

Since SAC is an off-policy algorithm it made sense to try and use this for the mission of reaching the goal. Taking advantage of the fact that it would do more ”random” move early on to try and explorer multiple ways to reach the goal before settling into a ”best” method.

During all of the training this has been somewhat of a cat and mouse game from designing a reward function to the model finding loophole to break it. While it showed early on that a method of just a very simple reward for gaining distance towards the goal showed great results, but due to time and computing limitations, this did not seem possible for our custom build. So with more complex rewards functions comes the problem of the model to find ways to exploit rewards to do other things than intended for better rewards. Some examples of this have been the robot sitting down on the back to still get upright rewards. While dodging the height requirement for ending an episode and gaining rewards for surviving more time steps. Another example is the robot throwing himself forward in a front flip motion to gain a large reward from velocity rewards. Finding a good balance of not to simple rewards so the model takes a long time to train but not to complex rewards so it starts exploiting it has been on of the greater challenges in this project.

While training has been running, we set Tensorboard logging for the SAC model since the run time would be up to several hours. To understand what is happening with the model, both the training model and the evaluation model could be monitor to either be able to stop training if it did not show any progress over a longer time or to be able to see if it was exploiting some rewards. A run could be seen in fig 2.

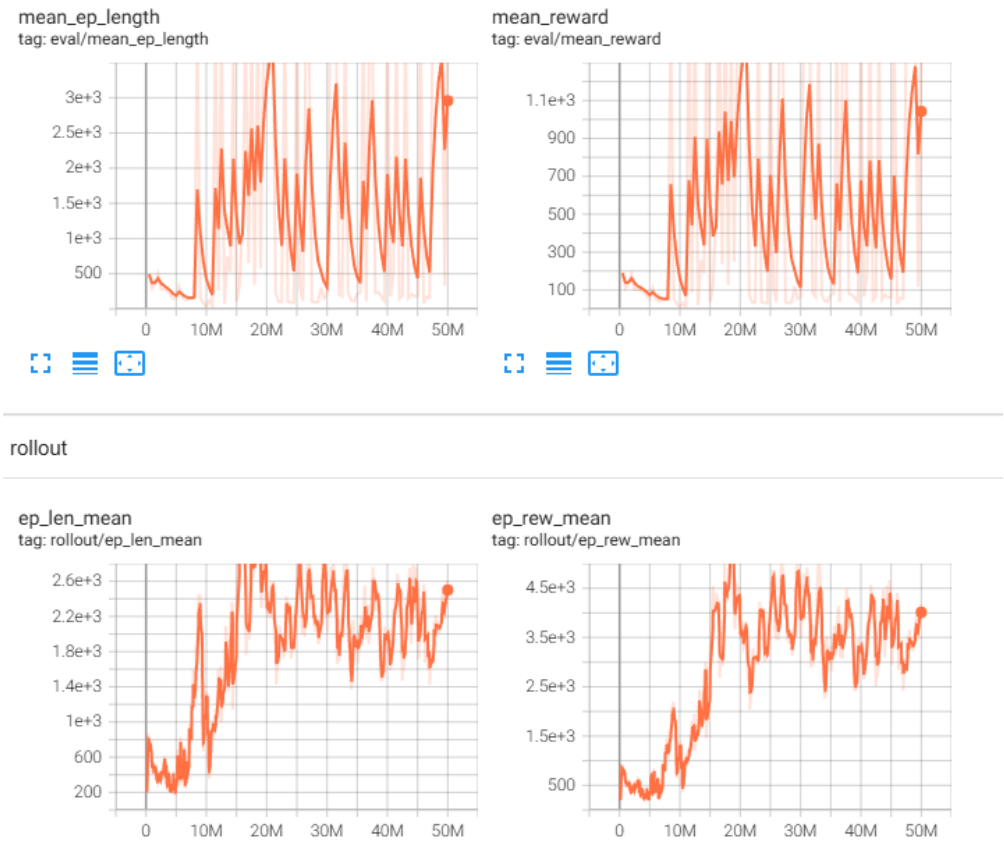

Figure 2: Tensorboard for best SAC model

6 Conclusion

In this project, Proximal Policy Optimization (PPO) and Soft Action-Critic (SAC) were implemented to train a simulated quadruped robot to walk. Due to time limits and resources, the robot did not learn to walk in a proper way. Two main outcomes were achieved, the PPO model that successfully learns to balance, while the SAC model developed a ”seal-like” crawling motion that allowed it to move towards the target. The main challenge throughout the project was to identify suitable hyperparameters and reward functions that effectively guided the learning process. Although the robot demonstrated basic balance and goal directed movement, further improvements are necessary for dynamic locomotion such as walking. Future work could focus on optimizing the weights in the reward function or in the reward function as a whole. For a more real life walking style the reward function should be adapted to limit the way the agent is allowed to move. This could become a quite large reward function to integrate this together with the goal of walking towards a target. This further extends the problem this paper had with the adjustment of the weights

in the reward functions to not have the agent exploit any rewards.

References

[1] Erwin Coumans and Yunfei Bai. PyBullet, a python module for physics simulation for games, robotics and machine learning. https://pypi.org/project/pybullet/, 2021.

[2] Stable Baselines3 Developers. Ppo — stable baselines3 documentation. https://stable-baselines3.readthedocs.io/en/master/modules/ppo.html, 2025. Accessed: 2025-10-21.

[3] Thomas Dudzik, Matthew Chignoli, Gerardo Bledt, Bryan Lim, Adam Miller, Donghyun Kim, and Sangbae Kim. Robust autonomous navigation of a small-scale quadruped robot in real-world environments. In *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3664–3671, 2020.

[4] Emma M. A. Harrison. Evaluating reinforcement learning algorithms for navigation in simulated robotic quadrupeds: A comparative study inspired by guide dog behaviour. https://arxiv.org/abs/2507.13277, 2025. arXiv preprint arXiv:2507.13277.

[5] OpenAI. Proximal policy optimization — spinning up. https://spinningup.openai.com/en/latest/algorithms/ppo.html, 2018.

[6] OpenAI. Soft actor-critic. https://spinningup.openai.com/en/latest/algorithms/sac.html, 2018. Accessed: 2025-10-20.

[7] Unitree Robotics. https://github.com/unitreerobotics, 2021. Accessed: 2025-10-20.

[8] ScienceDirect. Quadruped robot - an overview, 2025. Accessed: 2025-10-15.

[9] Qifeng Wan, Aocheng Luo, Yan Meng, Chong Zhang, Wanchao Chi, Shenghao Zhang, Yuzhen Liu, Qiuguo Zhu, Shihan Kong, and Junzhi Yu. Learning and reusing quadruped robot movement skills from biological dogs for higher-level tasks. *Sensors*, 24(1), 2024.

[10] Yilin Zhang, Jiayu Zeng, Huimin Sun, Honglin Sun, and Kenji Hashimoto. Dual-layer reinforcement learning for quadruped robot locomotion and speed control in complex environments. *Applied Sciences*, 14(19), 2024.