Reward Structures for Robotic Locomotion Tasks using Proximal Policy Optimization

Gabriel James Tidy
School of Computer Science & Applied Mathematics
University of the Witwatersrand
Johannesburg, South Africa
gabelancaster0@gmail.com

Benjamin Rosman (Supervisor)
School of Computer Science & Applied Mathematics
University of the Witwatersrand
Johannesburg, South Africa
benjamin.rosman1@wits.ac.za

***Abstract*—The use of robotic locomotion as a substitute for humans in practical tasks is important and wide-spread. Achieving robotic locomotion is generally only feasible through learning locomotion skills using Reinforcement Learning techniques, which require carefully designed and implemented reward structures. However, little research has been done on reward structures for robotic locomotion, especially those using legged robots. This paper designs and implements several reward structures using the popular Proximal Policy Optimization algorithm, for robotic locomotion focusing on speed, robot safety, or some combination of the two. Robot safety is measured using a novel metric introduced in this paper called the risk score. Some of the reward structures implemented are able to achieve safer locomotion compared to the default reward used in the OpenAI Gym training environment. In addition, it was found that behaviours with lower risk scores can potentially allow for faster, more stable locomotion, contrary to the tradeoff between speed and safety that was originally expected.**

***Index Terms*—reinforcement learning, robotics, locomotion, reward functions**

I. Introduction

The use of mobile robots is an important aspect of many fields today, from transportation to surveillance to emergency rescue operations. The key use of mobile robots is as a substitute for humans in tasks that are significantly dangerous, unsanitary, or repetitive. The vast majority of these tasks require the robot to be mobile: to be able to reach a multitude of positions in their environment [Llopis-Albert *et al.* 2019]. This paper focuses on the locomotive aspect of these tasks, specifically how to train robots to coordinate their actions to achieve locomotion.

The core problem for the field of robotic locomotion is developing systems to control the movement of the, perhaps, extremely complex robots consisting of many joints and moving parts. These control systems are often manually developed, however this process can be extremely time-consuming and often impossible to achieve correctly [Kohl and Stone 2004].

In order to reduce complexity and increase generalization, a process to learn the control system can be used instead. This process should be applicable to many different types of robots in many different types of environments. A popular method to achieve this is Reinforcement Learning (RL), where a robot can learn how to coordinate its actions to achieve locomotion through trial-and-error [Mosavi and Varkonyi 2017].

The main driving force for learning in RL tasks is the reward function, a feedback mechanism specified by the designer. This feedback takes the form of a positive or negative reward (penalty) given to the learning agent based on its chosen actions. Using the received reward, the agent develops an understanding of what actions it should and should not be taking in certain states of the environment [Icarte *et al.* 2022].

Due to this inherent importance of the supplied reward for learning, the design of the reward must be accurate for the desired task, as well as specific to the agent and its environment. The design of the reward, called the reward structure, determines how often rewards are given to the agent, what actions or environmental states lead to certain rewards, and the strength of these rewards (both positive and negative) [Cheng *et al.* 2022].

This culminates in the need for designers to ensure the reward structure used for a task leads to the learning of all desired behaviours associated with completing the task. The use of an incorrect reward structure will at best hamper the speed of learning, and at worst cause suboptimal or even completely incorrect behaviours to be learned [Icarte *et al.* 2022].

This paper focus specifically on Proximal Policy Optimization (PPO), a Reinforcement Learning method proposed by Dhariwal *et al.* [2017]. The state-of-the-art PPO algorithm is widely used, especially in the field of robotic locomotion [Kobayashi 2021]. However, little research has been done on locomotion-based reward structures, particularly for quadrupedal robots [Cheng *et al.* 2022].

This paper designs and implements several reward structures for PPO, specifically those structures relating to locomotion. Reward structures focusing on speed, safety, and some combinations of these factors are considered. Movement that is fast is useful in the majority of robotic locomotion tasks, as the faster a robot is able to move, the faster it is able to complete its assigned tasks. Fast movements in practical tasks, however, can sometimes lead to safety risks and potential damage to the robot itself [Eder *et al.* 2014].

Locomotion speed can be easily rewarded, as most robotic agents or environments provide a way to measure the velocity of the robot itself [Brockman *et al.* 2016]. The robot's joint-usage and likelihood of flipping over or crashing are good

indicators for robot safety, and thus can be used as the basis for negative rewards given to the robot to train for more cautious locomotion [Eder *et al.* 2014].

The experiments in this paper consist of training a quadrupedal robotic agent in a simulated environment, OpenAI Gym (developed by [Brockman *et al.* 2016]). The agent is trained multiple times using each custom-designed reward structure, and evaluated on its average speed and risk score, a metric designed in this paper for evaluating robot safety.

Although none of the reward functions implemented in this paper are able to surpass the default reward provided by the training environment in terms of speed, some of them are able to achieve safer locomotion. In addition, the results indicate that a lower risk score is not necessarily a tradeoff for higher movement speeds, as was originally expected, and safer locomotion can potentially allow for more stable, faster movement.

The remainder of this paper consists of background information related to this problem, such as robotic locomotion and Reinforcement Learning, specifically PPO (Section II), works related to this paper, specifically other robotics reward shaping techniques (Section III), the methodology used to implement and analyse the reward functions used (Section IV), the setup of the experiments run (Section V), the experiment results, along with a discussion thereof (Section VI) and finally a conclusion of this paper VII.

II. Background

A. Introduction

This section contains background information needed to understand the implementation of this paper. Specifically, this entails robotic locomotion and how it can be learned using Reinforcement Learning (Section II-B), as well as technical details for implementing the Reinforcement Learning methods used in this paper, specifically Proximal Policy Optimization (Section II-E).

B. Locomotion Skills for Complex Agents

Achieving fast, efficient and reliable locomotion on complex agents, such as robots, is a challenging task, as all of the robot's movements need to be precisely timed and accurately executed. These skills are often hand-coded, however this approach can be time-consuming and difficult to achieve correctly (as well as requiring updating whenever the robot or its environment changes) [Kohl and Stone 2004].

Instead, these locomotion skills can be learned using Reinforcement Learning techniques such as Actor-Critic Policy Gradient methods (detailed in Section II-D). This allows the robot to learn in a variety of environments, and potentially learn a variety of locomotion styles, or gaits. With the correct implementation, the learned gait is also very likely to have favourable results compared to hand-coded gaits, and may even surpass them [Kohl and Stone 2004].

In this paper robots that achieve locomotion using 'legs', made up of a series of joints, are considered. The actions available to the robotic agent are the modification of these joint angles, allowing locomotion to be achieved. Reinforcement Learning for these robotic agents consists of learning a policy to control their joint angles in response to the current environmental state (typically consisting of the current joint angles and other information detailing the status and positioning of the robot) [Carpentier and Wieber 2021].

Legged robots can come in a variety of forms, with differing body structures, numbers of legs, and degrees of freedom for motion. In this paper specifically a four-legged (quadrupedal) robot with two links and joints per leg is used, however the underlying techniques remain the same for all legged robots, and it has been shown that locomotion learning algorithms can be easily adapted to work for varying motion structures (Section [Carpentier and Wieber 2021]).

C. Reinforcement Learning

Reinforcement Learning (RL) is a branch of machine learning that uses trial-and-error through repeated interaction with the environment [Nasteski 2017]. Learning occurs on the basis of a numerical feedback signal (reward or penalty) based on the performance at a given time step to influence future learning [Bratko *et al.* 1998]. This is accomplished by learning a policy that maps from environmental states to possible actions. Actions will in some way affect the environment, and based on this, some feedback, called the reward, can be applied to the policy [Nasteski 2017].

Although technically a form of supervised learning, reinforcement learning differs in that there are no input/output pairs presented to the algorithm. Instead, after making a certain action, the algorithm is given feedback in the form of the resulting environmental state and the associated reward (possibly negative). As opposed to traditional supervised learning techniques, the algorithm is not told which action would have been best, and must rather gather experience through trial-and-error to determine the best actions in each state [Kaelbling *et al.* 1996].

1) Markov Decision Processes: Formally, RL uses a framework known as Markov Decision Processes (MDPs). MDPs consist of states, actions, transitions and rewards, and are defined as a tuple $\langle S, A, T, R \rangle$, with each element explained below: [van Otterlo and Wiering 2012]:

- **State** - a (unique) characterization of the environment, including everything needed to represent any differences that may occur in the environment. States are used to provide information to the learning agent, so that they may choose their actions based on the current environment. States are represented by S, the set of all possible (unique) configurations of the environment.
- **Action** - anything the learning agent is able to do that changes the state of the environment. Actions represent the behaviour of the agent and are what RL tasks focus on learning. Actions are represented by A, the set of all possible (unique) actions available to the learning agent.
- **Transition** - modelled by a *transition function*, this determines how states change in response to actions. Formally, a transition function T is defined as T :

$S \times A \times S \rightarrow [0, 1]$, such that $T(s, a, s')$ is the probability of transitioning to state s' from state s after taking action a.

- **Reward** - the feedback signal provided to the learning agent based on its chosen actions. This is modelled using a *reward function* R, where $R : S \times A \times S \rightarrow \mathbb{R}$ such that $R(s, a, s')$ is the reward received after taking action a in state s and transitioning to state s'. Simpler forms of the reward function can also be used, such as $R : S \rightarrow \mathbb{R}$ and $R : S \times A \rightarrow \mathbb{R}$, where rewards are given solely based on the current state or the current state and chosen action, respectively.

2) Policies: A policy is a mapping from states to actions and controls the behaviour of the learned agent. Formally, a policy π can be deterministic, where $\pi : S \rightarrow A$ such that $\pi(s)$ gives the action to perform in state s, or stochastic, where $\pi : S \times A \rightarrow [0, 1]$ such that $\pi(s, a)$ gives the probability of performing action a in state s [van Otterlo and Wiering 2012].

The policy is not part of the MDP, but rather an aspect of the agent itself, and the learning of the policy mapping is the core problem of RL [van Otterlo and Wiering 2012].

3) Returns: Optimizing the policy in terms of the immediate rewards produces behaviour that is purely myopic. However, rewards in MDPs are typically delayed, meaning rewards for performing an action or series of actions may only be received long after they were taken. Therefore, RL problems instead typically consider *future rewards*, called returns. Formally, the return at a specific time step is defined as [van Otterlo and Wiering 2012]:

$$G_t = \sum_{t=1}^{T} \gamma^{t-1} R_t$$

where

- t is the current time step.
- T is the number of time steps considered. This could be infinite for continuing tasks, or finite for episodic tasks.
- R_t is the reward received at time step t.
- γ is the discount factor, where $\gamma \in [0, 1]$. Future rewards are discounted ($\gamma < 1$) specifically for continuing tasks to avoid infinite sums, or generally to encourage receiving rewards as soon as possible. Note that if $\gamma = 1$, no discounting is performed and if $\gamma = 0$, returns are purely myopic (only the immediate reward is considered).

4) Value functions: The true value of a return for a given time step is often difficult or impossible to compute, given the temporal nature of RL problems. To overcome this, the concept of *expected returns* is introduced. The expected return is an approximation of the true return, and calculating it does not require running a policy in an environment for any further time steps past the current one [van Otterlo and Wiering 2012].

Expected returns are used in *value functions*. A value function is an estimate of the quality, or value, of a given state. In particular, there are two forms of value functions, the state value function and the state-action value function [van Otterlo and Wiering 2012]. This paper makes use of the *state value function* $V^{\pi}(s)$, which is the expected return of following policy π starting from state s, defined as

$$V^{\pi}(s) = \sum_{s' \in S} T(s, \pi(s), s')(R(s, \pi(s), s') + \gamma V^{\pi}(s'))$$

The core RL problem is finding the *optimal policy* $\pi^*(s)$ which gives the best action to perform in state s with respect to the MDP, defined such that: [van Otterlo and Wiering 2012]

$$\forall s \in S, \forall \text{ policies } \pi, V^{\pi^*}(s) \geq V^{\pi}(s)$$

where $V^{\pi^*}(s) = V^*(s)$ is the *optimal state value function* defined as:

$$V^*(s) = \max_{a \in A} \sum_{s' \in S} T(s, a, s')(R(s, a, s') + \gamma V^*(s'))$$

The optimal state value function is called the *Bellman optimality equation*, and its intuitive meaning is that the value of state s under optimal policy π^* is equal to the expected return achieved by taking the best action in state s [van Otterlo and Wiering 2012].

If the optimal state value function $V^*(s)$ is known (either from being learned or directly computed), it can be used to determine the optimal policy $\pi^*(s)$ with the following equation: [van Otterlo and Wiering 2012]

$$\pi^*(s) = \arg\max_{a} \sum_{s' \in S} T(s, a, s')(R(s, a, s') + \gamma V^*(s'))$$

5) Reward Structures: The use of reward shaping is imperative to achieving accurate learning for RL tasks, as using incorrect reward functions could not only cause learning speed to be significantly hampered, but can also cause learning to not occur at all (or for the incorrect tasks/behaviours to be learned) [Cheng *et al.* 2022].

For this reason, a good understanding of the task and required reward structures is necessary for any RL problem, as the agent's learning is directly tied to the feedback it receives (the reward) [Cheng *et al.* 2022]. As such, this paper provides an analysis of the different reward structures that can be used for robotic locomotive tasks, and the different behaviours they generate.

D. Policy Gradient Methods

Policy Gradient methods are a type of Model-free Reinforcement Learning algorithm. Not having access to the model means not having access to the transition dynamics T or the reward function R. This means that the Bellman optimality equation cannot be computed and used to determine the optimal policy π^* [van Otterlo and Wiering 2012].

Instead, Policy Gradient methods make use of gradient ascent to optimize a policy (mapping between actions and states) with respect to its expected rewards [Mansour *et al.* 1999].

Gradient ascent is an optimization technique that maximizes the value of an objective function. This is accomplished by moving the objective function parameters in the direction of greatest increase of the function value [Ruder 2016]. In

the case of Policy Gradient methods, the parameters are the policy parameters and the objective function is the long-term expected reward function [Mansour *et al.* 1999].

The typical objective function for Policy Gradient methods is [Dhariwal *et al.* 2017]:

$$J(\pi_\theta) = \hat{\mathbb{E}}_t[\log \pi_\theta(a_t|s_t)\hat{A}_t] \tag{1}$$

where

- t is the current time step
- θ is the policy parameters / weights
- s_t is the state at time step t
- a_t is the chosen action at time step t
- π_θ is the policy using parameters θ
- $\hat{A}_t$ is the estimator of the future reward at time step t, for example $V^{\pi_\theta}(s_t)$
- $\hat{\mathbb{E}}_t$ is the expectation function, which indicates the average over a finite number of samples

In order to determine the direction of greatest increase of the objective function, its gradient $\nabla_\theta J(\pi_\theta)$ is needed, where [Degris *et al.* 2014]

$$\nabla_\theta J(\pi_\theta) = \hat{\mathbb{E}}_t[\nabla_\theta \log \pi_\theta(a_t|s_t)\hat{A}_t]$$

If updates are being performed for a single time step, the expectation $\hat{E}_t$ can be removed and the reward estimator $\hat{A}_t$ can be considered the true reward for that time step, R_t [Degris *et al.* 2014].

Once the gradient of the objective function is computed with respect to the current policy's parameters, it can be used to update the parameters according to the equation: [Degris *et al.* 2014]

$$\theta_{new} = \theta_{old} + \alpha \nabla_\theta J(\pi_\theta)$$

where α is the learning rate, affecting the step size of the update.

1) Actor-Critic Methods: Actor-Critic algorithms are a type of Policy Gradient method where instead of solely learning a policy, a value function is also learned using Model-Free RL techniques. Value functions are used to learn the value of particular states, and help guide the agent in choosing between them.

Pure Policy Gradient methods (actor-only methods) typically suffer from large amounts of variance while learning. Actor-Critic methods aim to overcome this by updating the actor (policy) using information learned by the critic (value function) [Tsitsiklis and Konda 1999].

When using a critic, the estimate of the future rewards $\hat{A}_t$ in Equation 1 is not computed using the true value function for the current policy, such as $V^{\pi_\theta}(s_t)$. Instead, an estimator of this true value function parameterized by ω is used, namely $V_\omega(s_t)$ [Degris *et al.* 2014].

This parameterized value function is learned alongside the policy π_θ, and the update equations become: [Degris *et al.* 2014]

$$\theta_{new} = \theta_{old} + \alpha_\theta \delta_t \nabla_\theta \log \pi_\theta(a_t|s_t) \tag{2}$$

$$\omega_{new} = \omega_{old} + \alpha_\omega \delta_t \nabla_\omega V_\omega(s_t) \tag{3}$$

where

- t is the current time step.
- α_θ and α_ω are the actor and critic learning rates, respectively.
- δ_t is the critic error at time step t defined as:

$$\delta_t = R_t + \gamma V^\omega(s_{t+1}) - V_\omega(s_t) \tag{4}$$

- s_t is the current state at time step t.
- a_t is the action chosen at the current state, with s_{t+1} being the resulting state at the next time step $t+1$.

E. Proximal Policy Optimization

Proximal Policy Optimization (PPO) is a family of Policy Gradient Reinforcement Learning methods and was presented by Dhariwal *et al.* [2017], who found that PPO generally outperformed other state-of-the-art policy gradient methods in terms of training time, simplicity, and final agent performance.

For this paper, the method using the clipped surrogate objective function is considered. This works the same as regular Policy Gradient methods, except the typical objective function given in Equation 1 is replaced with a novel objective function that used clipped probability ratios. This function is a pessimistic estimate for the policy performance, and is given by [Dhariwal *et al.* 2017]:

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t[\min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)] \tag{5}$$

where

- $r_t(\theta)$ is the probability ratio for the policy, given by:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$$

- ϵ is a hyperparameter affecting how much clipping is performed, eg. $\epsilon = 0.2$

Inside this objective function, the minimum is taken between the unclipped objective (first term), and the clipped objective (second term). This means the final objective is a lower bound of the unclipped objective. The clipped objective removes the incentive of r_t to move outside the range $[1-\epsilon, 1+\epsilon]$. By doing this, it is ensured that the change in probability ratios is only included when it would make the objective function worse, and ignored when it would improve the objective (as desired) [Dhariwal *et al.* 2017].

PPO makes use of Generalized Advantage Estimation (GAE) first introduced by Abbeel *et al.* [2015] in order to calculate an estimate of the future rewards (called the advantage estimate $\hat{A}$) from a given time step, which is used in the calculation of the loss value [Dhariwal *et al.* 2017].

The definition of the advantage estimate for a single time step using GAE is given by Abbeel *et al.* [2015] as:

$$\hat{A}_t(\gamma, \lambda) = \sum_{k=0}^{T} = (\gamma\lambda)^k \delta_{t+k} \tag{6}$$

where

- t is the time step that the advantage estimate is being computed for, and is the starting point of the calculation.

- T is the number of time steps considered. This could be infinite for continuing tasks, or finite for episodic tasks.
- γ and λ are hyperparameter discount factors. The specific values used in this paper are given in Section V-C.
- δ_{t+k} is the critic error at time step $t+k$, as defined in Equation 4.

This paper makes specific use of the Actor-Critic version of the PPO algorithm from Dhariwal *et al.* [2017], which includes both policy and value function learning, as detailed in Section II-D1. When using a critic, in addition to the actor loss determined by Equation 5, the loss incurred by the critic must also be considered. The critic loss is defined as: [Dhariwal *et al.* 2017]

$$L^{Critic}(\omega) = \hat{\mathbb{E}}_t(\hat{A}_t - V^{\omega}(s_t))^2$$

where

- $\hat{A}_t$ is the advantage estimate of time step t, computed using GAE (Equation 6).
- $V^{\omega}(s_t)$ is the value estimate of the state s_t given by the critic.

This gives the calculation of the total loss to be:

$$L^{Total}(\theta, \omega) = L^{CLIP}(\theta) + W_{critic}L^{Critic}(\omega) - W_e\hat{\mathbb{E}}_t[e_t] \tag{7}$$

where

- W_{critic} and W_e are hyperparameter weights affecting the relative strength of the critic and entropy losses, respectively. The specific values used in this paper are given in Section V-C.
- e_t is the entropy incurred by the actor at time step t. Entropy is a measure of the uncertainty over a probability distribution [Cover and van Campenhout 1981]. In this case, it is a measure over the probability distribution of the joint-values supplied by the actor. Distributions with higher standard deviations, and thus higher uncertainties, will produce high entropy values. Adding the negative entropy to the loss value will thus encourage more stable distributions with lower standard deviations [Dhariwal *et al.* 2017].

1) Neural Networks: Neural networks are used for both the actor and critic networks of the PPO algorithm. The networks take as inputs the observations of the agent at each time step, with the critic outputting the state value for that time step, and the actor outputting a Normal distribution for each joint of the agent.

The joint action values are sampled from this distribution for each time step during training, however, during model testing deterministic values are used (the means of each distribution). The main actor network outputs the mean values for the distribution (one mean for each joint), while the standard deviations for the distribution are determined using a separate neural parameter calculated using the main actor network's output.

2) PPO Pseudocode: Training a policy π_θ (actor) and value function V_ω (critic) using PPO from Dhariwal *et al.* [2017] consists of running the actor and critic in the environment for a number of time steps, T, to generate a trajectory. This trajectory consists of:

- states for each time step, from the environment.
- state values generated by the critic for each time step.
- action distributions generated by the actor for each time step, along with the sampled actions for that time step.
- rewards received for each time step.

From the rewards and state values in the trajectory, the advantage estimation for each time step is then computed using GAE (Equation 6). A number of update epochs, N, are then run on the actor and critic weights using the trajectory and advantage estimations. For each update epoch, the trajectory is split into M mini-batches of fixed length. From these mini-batches, the total loss is calculated using Equation 7. The gradient of this loss is used to update the weights for the actor and critic using Equations 2 and 3, respectively.

Once each update epoch has finished, the actor and critic weights are reset to their updated values, and a new trajectory is generated. This process is repeated until convergence of the actor and critic weights is reached.

This process is shown in the pseudocode in Algorithm 1. The hyperparameters present in the pseudocode are the trajectory length T, the number of update epochs N, the mini-batch size determining M as well as the weights for the total loss calculation L^{total}. The values used in this paper for these hyperparameters are detailed in Section V-C.

Algorithm 1 PPO

for $iteration = 1, 2, \ldots$ **do**
  Generate trajectory of length T from the environment using actor π_θ and critic V_ω
  Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$ using Equation 6
  for $m = 1, 2, \ldots, N$ **do**
    for $m = 1, 2, \ldots, M$ **do**
      Sample mini-batch m from trajectory
      Calculate total loss L^{total} using Equation 7
      Perform gradient descent on L^{total} wrt. actor weights θ and critic weights ω to get updated weights θ_{new} and ω_{new} using Equations 2 and 3
    end for
  end for
  $\theta \leftarrow \theta_{new}$
  $\omega \leftarrow \omega_{new}$
end for

III. Related Works

A. Reward Shaping for Robotic Tasks

The effectiveness of Reinforcement Learning for robotics tasks rely heavily on the accuracy of the reward function, however in real-world applications the environments or observation tools of the agent (such as kinetic sensors) are often noisy, leading to uncertainty affecting learning [Cheng *et al.* 2022].

Reward shaping typically falls under one of two main categories: sparse rewards or dense rewards. Sparse rewards give feedback only when a task (or sub-task) has been completed, whether successful or not. This typically leads to slow training times, as large amounts of data needs to be collected to overcome the low amounts of gradient information relating to the reward signal [Cheng *et al.* 2022].

Dense rewards, on the other hand, aim to overcome these sample efficiency issues by providing continuous feedback to the agent to accelerate the learning process. Although dense rewards do typically allow faster learning, they are highly dependent on the accuracy of the reward signal [Cheng *et al.* 2022].

Cheng *et al.* [2022] aimed to combine the advantages of both sparse and dense rewards by implementing a method called Dense2Sparse, where the agent initially receives dense rewards in order to quickly learn a sub-optimal policy, then switches to receiving sparse rewards which typically are less affected by noise in real-world applications. Once training on sparse rewards, the agent is able to still quickly learn an optimal policy as the data it is collecting is of higher quality, due to already following a near-optimal policy [Cheng *et al.* 2022].

This paper uses ideas from Cheng *et al.* [2022], although a simulated environment is used which contains much less noise than a real-world environment. For this reason, dense rewards are used to provide continuous feedback to the agent in order to have quick learning. In addition, the task itself (forward robotic locomotion) is continuous and does not contain completion states, making sparse rewards ineffective.

IV. Methodology

A. Reward Functions

A total of 8 different reward functions are considered in this paper (along with the default reward provided by the simulation environment from Section V-A). Each reward function has as a base metric the forward-velocity of the agent (positive velocity yields a positive reward, and a negative velocity indicating backwards movement yields a negative reward). This is the main training feedback for forward locomotion, and as such is used for all reward functions to train for the speed component of locomotion.

The reward functions implemented in this paper are separated into *primitive rewards* and *combined rewards*. Primitive rewards consider only the forward-velocity of the agent and one or zero additional metrics. Combined rewards consider the forward-velocity of the agent along with two or three additional metrics used in the primitive rewards.

In the below definitions, the following terms are used:

- t - the current time step.
- W_v - the weight applied to the forward-velocity of the agent.
- W_c - the weight applied to the joint-usage of the agent.
- v_t^f - the forward-velocity of the agent at time step t.
- a_t^j - the angle of joint j of the agent at time step t.
- J - the number of joints controllable by the agent.
- R_f - the (negative) reward associated with the agent flipping upside-down.
- R_s - the (positive) reward associated with the agent not flipping upside-down.

The terms W_v, W_c, R_f and R_s are hyperparameters with the specific values used in this paper given in Section V-C. The terms v_t^f, a_t^j and the information used to determine when the agent has flipped upside-down are provided by the simulation environment at each time step as part of the environment state at that time step, as detailed in Section V-A.

1) Primitive rewards: The primitive rewards are defined and implemented as follows:

- **Baseline reward** - applies a positive reward based on the forward-velocity of the agent:

$$R_t^B = W_v v_t^f$$

- **Conservative reward** - applies a positive reward based on the forward-velocity of the agent, and a negative reward based on the amount of joint-usage of the agent:

$$R_t^C = R_t^B - W_c \sum_{j=1}^{J} a_t^j$$

- **Scared reward** - applies a positive reward based on the forward-velocity of the agent, and a massive negative reward whenever the agent flips upside-down:

$$R_t^{Sc} = \begin{cases} R_t^B + R_f, & \text{if agent has flipped} \\ R_t^B, & \text{otherwise} \end{cases}$$

- **Survivor reward** - applies a positive reward based on the forward-velocity of the agent, and a small positive reward for each time step the agent is not flipped upside-down:

$$R_t^{Su} = \begin{cases} R_t^B, & \text{if agent has flipped} \\ R_t^B + R_s, & \text{otherwise} \end{cases}$$

Joint-usage is a common metric used in Reinforcement Learning robotics-tasks, as it is both easily available to the agent and a good indicator of extreme movement, which could be dangerous to the robot [Eder *et al.* 2014]. The agent flipping upside-down is not desirable for two reasons; the agent cannot make forward progress while upside down, and flipping is likely to cause damage to the robot itself. Penalising the agent for flipping (or rewarding the agent for each time step it is not flipped) is thus clearly useful when training for robot safety as well as movement speed.

It is expected that the Baseline reward will produce locomotion focusing mainly on speed with little concern for safety, as it has no metrics to reward safe, or penalize risky, behaviour. However, the Conservative, Scared and Survivor rewards all include some metric to reward safe, or penalize risky, behaviour, and as such are expected to produce locomotion that is safer, although perhaps slower, than that of the Baseline reward.

Of these three 'risk averse' rewards, it is expected that the Conservative reward will be the safest and slowest, as

penalizing joint-usage is the most invasive metric used. On the contrary, the Scared reward is expected to produce the fastest movement out of these three, as it only penalizes the agent when it flips upside-down, and otherwise does not restrict its actions. The Survivor reward is expected to make the least progress while learning, as the structure of rewarding the agent for each time step it is not flipped upside-down has the potential to encourage behaviour where the agent remains motionless and still accrues positive rewards.

2) Combined rewards: The combined rewards are defined and implemented as follows:

- **Conservative-Scared reward** - applies a positive reward based on the forward-velocity of the agent, a negative reward based on the amount of joint-usage of the agent, and a massive negative reward whenever the agent flips upside-down:

$$R_t^{CSc} = \begin{cases} R_t^B - W_c \sum_{j=1}^{J} a_t^j + R_f, & \text{if agent has flipped} \\ R_t^B - W_c \sum_{j=1}^{J} a_t^j, & \text{otherwise} \end{cases}$$

- **Conservative-Survivor reward** - applies a positive reward based on the forward-velocity of the agent, a negative reward based on the amount of joint-usage of the agent, and a small positive reward for each time step the agent is not flipped upside-down:

$$R_t^{CSu} = \begin{cases} R_t^B - W_c \sum_{j=1}^{J} a_t^j, & \text{if agent has flipped} \\ R_t^B - W_c \sum_{j=1}^{J} a_t^j + R_s, & \text{otherwise} \end{cases}$$

- **Scared-Survivor reward** - applies a positive reward based on the forward-velocity of the agent, a massive negative reward whenever the agent flips upside-down, and a small positive reward for each time step the agent is not flipped upside-down:

$$R_t^{ScSu} = \begin{cases} R_t^B + R_f, & \text{if agent has flipped} \\ R_t^B + R_s, & \text{otherwise} \end{cases}$$

- **Conservative-Scared-Survivor reward** - applies a positive reward based on the forward-velocity of the agent, a negative reward based on the amount of joint-usage of the agent, a massive negative reward whenever the agent flips upside-down, and a small positive reward for each time step the agent is not flipped upside-down:

$$\mathrm{R}_t^{CScSu} = \begin{cases} R_t^B - W_c \sum_{j=1}^{J} a_t^j + R_f, & \text{if agent has flipped} \\ R_t^B - W_c \sum_{j=1}^{J} a_t^j + R_s, & \text{otherwise} \end{cases}$$

Of the four combined rewards, it is expected that they will all produce locomotion that is slower but safer compared to the locomotion of the four primitive rewards, as they provide additional rewards / penalties for safe / risky behaviour, respectively. In particular, the Conservative-Scared-Survivor reward is expected to produce the safest but slowest locomotion due to it using the most metrics encouraging safety out of all implemented rewards.

B. Evaluation Metrics

During training, the agent is rewarded using one of the reward functions detailed in Section IV-A. The agent is also evaluated using each other reward function to provide comparisons discussed in Section VI-A (however these rewards are not available to the agent itself).

In order to evaluate the performance of each reward function for the locomotion aspects of speed and robot safety, two additional metrics are used:

- Movement speed is evaluated using the average forward-velocity of the agent measured across multiple episodes. (Higher movement speed values are better.)
- Robot safety is evaluated using the **risk score** (lower risk score values are better). This is calculated by taking the average between two metrics:
  - The average vertical-velocity of the agent measured across multiple episodes.
  - The average angular velocity of the agent's torso measured across multiple episodes. The angular velocity is measured at each time step as the average between the angular velocities in the x-, y- and z-dimensions.

High values of the agent's vertical-velocity indicate either rapid movements up and down, or the agent jumping high into the air, and subsequently falling down again. These both pose risks towards the safety of the robot and are thus used to calculate the risk score. Likewise, high values of the agent's angular-velocities indicate the agent rapidly twisting or turning, making the agent more prone to flipping upside-down.

V. Experiments Setup

A. Training Domain

The training environment was chosen as the OpenAI Gym, as it is an industry standard for simulated reinforcement learning tasks. OpenAI Gym is an open-source Python library providing multiple simulated environments and agents, including simple and complex robotic agents. [Brockman *et al.* 2016].

The specific Gym environment used is the MuJoCo environment, which is used for 2D and 3D simulated robotics tasks. Using the MuJoCo physics engine allows the simulation to run both accurately and quickly [Erez *et al.* 2012]. The MuJoCo environment chosen for this paper is a flat, empty plane with no obstacles. This should allow locomotion to be unhindered, and entirely reliant on the learned policy for the joints.

From the MuJoCo environment, the quadrupedal Ant robot is used as the training agent. This 3D robot is based on the work by Abbeel *et al.* [2015] and consists of a torso and four attached legs. The torso is a free rotational body, and each leg is made up of two links, connected by a joint. Each leg is also connected to the torso with a separate joint.

The Ant robot can observe the following from the environment:

- The height (z-coordinate) of the Ant's torso (the x- and y-coordinates are not included as they are used to

determine the reward function, and as such the policy should develop an abstract understanding of them from the rest of the observations [Abbeel *et al.* 2015]).
- The velocities of the torso in the x-, y- and z-direction.
- The orientation of the torso given as a four-dimensional quaternion representation (x, y, z and w). Quaternions are a method of representing the rotation of a three-dimensional body [Perumal 2014].
- The angular velocities of the torso (about the x-, y- and z-axis).
- The joint angles between the torso and each leg (hip angles), as well as their angular velocities.
- The joint angles between the upper and lower link of each leg (ankle angles), as well as their angular velocities.

The action taken by the Ant robot is determined by eight torque values, one for each joint in the robot. The goal is to learn a policy controlling the movement of each of the eight joints in order to produce forward locomotion (using the observations as inputs and learning from the rewards).

B. Neural Network Architecture

The simulated Ant robot from the MuJoCo Gym environment makes 27 total observations per time step, and the Ant's actions are determined by the torques applied to its eight joints (from section V-A). This means both the actor and critic neural networks will consist of input layers with 27 neurons (one for each observation). The critic network has a single output neuron for the state value, while the actor network has 8 output neurons, one for each joint (see Section II-E1 for an explanation).

In addition, each network contains a single hidden layer using the ReLU activation function, with the number of hidden neurons specified as a hyperparameter (see Section V-C). Only one hidden layer is used as the agent is still able to achieve stable learning using this relatively simple neural network, likely due to the low amount of observations from the environment, and joints in the agent. Using a more complicated agent or environment (such as one including obstacles) would likely require larger networks.

C. Training Hyperparameters

TABLE I
PPO HYPERPARAMETER VALUES

Hyperparameter	Value
α_θ	0.0001
α_ω	0.0001
γ	0.99
λ	0.95
T	8192
ϵ	0.2
W_{critic}	0.1
W_e	0.001
N	10
Mini-batch size	64
Hidden layer size	128

The PPO algorithm hyperparameters values used in this paper are shown in Table I, where:

- α_θ and α_ω are the actor and critic learning rates used in Equations 2 and 3, respectively.
- γ and λ are the discount factors used in the GAE calculation (Equation 6).
- T is the trajectory length used in the GAE calculation (Equation 6).
- ϵ is the PPO clipping factor used in Equation 5.
- W_{critic} and W_e are the weights affecting the relative strength of the critic and entropy losses, respectively. They are used in Equation 7.
- N is the number of PPO update epochs performed for each trajectory, seen in Algorithm 1.
- The mini-batch size is the number of time steps present in each mini-batch sampled from the trajectory. This size determines the number of mini-batches M, seen in Algorithm 1.
- The hidden layer size is the number of nodes used in the hidden layers of the actor and critic neural networks, explained in Section V-B.

The values chosen were taken as the suggested values from the original implementation by Dhariwal *et al.* [2017], with the exception of the trajectory length, hidden layer size and the weights for the loss terms. These values were hand-tuned until stable learning was observed using the Baseline reward from Section IV-A.

VI. RESULTS

To get the results, the agent was trained using each reward type from Section IV-A three times. Each of these runs was then tested ten times to determine their average movement speed and risk score (detailed in Section IV-B). These results are shown in Figure 1. Each dot in the figure represents one run trained using the associated reward type (with the metrics averaged over ten tests).

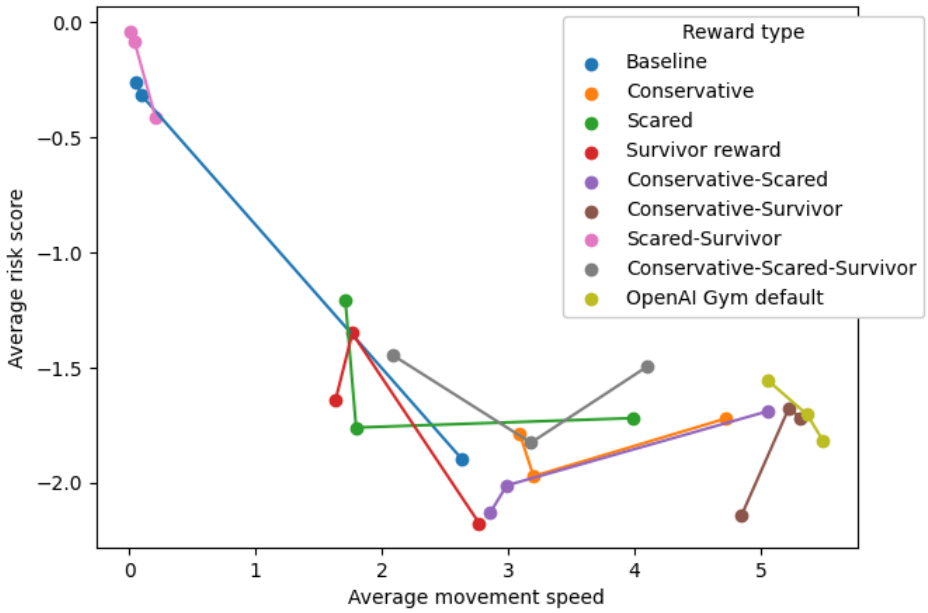
Additional plots for the agent's returns (Figure 2) and total loss (Figure 3) during training, as well as links to video recordings of the behaviours learned using the different reward functions, can be found in the Appendix (Section VIII).

A. Discussion

From the results in Figure 1, we can see the variety in behaviours across the different reward functions. The Scared-Survivor and Baseline rewards highly prioritize safety over speed, as seen by their extremely low risk scores and movement speed. In contrast, the other reward functions place more emphasis on movement speed at the sacrifice of their risk score. Of particular note is the Conservative-Survivor reward, which is able to achieve the best movement speed among all reward functions implemented in this paper.

In addition, from the total losses during training shown in Figure 3, it can be seen that learning actually occurs using most of the reward functions implemented, as their total loss was able to decrease to convergence. However, the two rewards with extremely low movement speeds, Baseline and

Fig. 1. Average speed vs risk across all reward types



Scared-Survivor, were not able to achieve adequate degrees of learning, as seen by their returns during training in Figure 2.

None of the reward functions were able to match the default reward implemented by the OpenAI Gym environment, which achieves the highest average speed on the results seen in Figure 1. This is likely due to the fact that the default reward is able to include metrics not available in the observations from the environment, namely the contact forces between the robot and the ground [Abbeel *et al.* 2015]. Using these contact forces, it is possible to directly penalize extreme movements of the robot, encouraging safe locomotion without discouraging fast locomotion.

Despite the inability to produce locomotion faster than the default reward, the reward functions implemented in this paper still could be useful if robot safety is a higher priority for the specific application. This is seen by the Scared and Survivor reward types yielding lower risk scores compared to the default reward, albeit at the cost of movement speed. The Scared-Survivor and Baseline rewards, despite having by far the lowest risk scores, would likely produce movement that is too slow to be useful in practical applications.

In general, the combined rewards were able to achieve faster movement than the primitive rewards, with the exception of the Scared-Survivor reward. This is unexpected, as the combined rewards were expected to produce safer yet slower locomotion due to their inclusion or additional metrics encouraging safer behaviour. It is possible that safer, more stable locomotion actually allows faster overall speeds, at least for certain rewards like the Conservative-Survivor reward, contrary to the trade-off between the two that was originally expected.

VII. Conclusion

This paper designed and implemented a number of reward structures for achieving locomotion in robotics tasks using Reinforcement Learning, specifically Proximal Policy Optimization. A novel metric for determining robot safety was also introduced, namely the risk score.

Despite none of the implemented rewards being able to match the speed of the default reward provided by the training environment, they were able to produce safer locomotion, and as such could be useful for practical applications where safety is of a higher priority.

In addition, it was discovered that a low risk score does not necessarily require low locomotion speeds, as was originally expected. Instead, it seems that lower risk behaviour has the potential to allow for faster, more stable movement.

References

[Abbeel *et al.* 2015] P. Abbeel, M. Jordan, S. Levine, P. Moritz, and J. Schulman. High-dimensional continuous control using generalized advantage estimation. In *arXiv preprint arXiv:1506.02438*, 2015.

[Bratko *et al.* 1998] I. Bratko, M. Kubat, and R. S. Michalski. A review of machine learning methods. In *Machine Learning and Data Mining: Methods and Applications*, pages 3–69, 1998.

[Brockman *et al.* 2016] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. Openai gym. In *arXiv:1606.01540*, 2016.

[Carpentier and Wieber 2021] J. Carpentier and P. B. Wieber. Recent progress in legged robots locomotion

control. In *Current Robotics Reports*, volume 2.3, pages 231–238, 2021.

[Cheng *et al.* 2022] E. Cheng, K. Dong, Y. Luo, B. Song, Z. Sun, Q. Zhang, L. Zhao, and C. Zhou. Balance between efficient and effective learning: Dense2sparse reward shaping for robot manipulation with environment uncertainty. In *IEEE/ASME International Conference on Advanced Intelligent Mechatronics (AIM)*, pages 1192–1198, 2022.

[Cover and van Campenhout 1981] T. Cover and J. van Campenhout. Maximum entropy and conditional probability. In *IEEE Transactions on Information Theory*, volume 27.4, pages 483–489, 1981.

[Degris *et al.* 2014] T. Degris, N. Heess, G. Lever, M. Riedmiller, D. Silver, and D. Wierstra. Deterministic policy gradient algorithms. In *International conference on machine learning*, pages 387–395, 2014.

[Dhariwal *et al.* 2017] P. Dhariwal, O. Klimov, A. Radford, J. Schulman, and F. Wolksi. Proximal Policy Optimization algorithms. In *arXiv preprint arXiv:1707.06347*, 2017.

[Eder *et al.* 2014] K. Eder, C. Harper, and U. Leonards. Towards the safety of human-in-the-loop robotics: Challenges and opportunities for safety assurance of robotic co-workers. In *IEEE International Symposium on Robot and Human Interactive Communication*, volume 23, pages 660–665, 2014.

[Erez *et al.* 2012] T. Erez, Y. Tassa, and E. Todorov. Mujoco: A physics engine for model-based control. In *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference*, pages 5026–5033, 2012.

[Icarte *et al.* 2022] R. T. Icarte, T. Q. Klassen, S. A. McIlraith, and R. Valenzano. Reward machines: Exploiting reward function structure in reinforcement learning. In *Journal of Artificial Intelligence Research*, volume 73, pages 173–208, 2022.

[Kaelbling *et al.* 1996] L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey. In *Journal of artificial intelligence research*, volume 4, pages 237–285, 1996.

[Kobayashi 2021] T. Kobayashi. Proximal policy optimization with relative pearson divergence. In *IEEE International Conference on Robotics and Automation (ICRA)*, pages 8416–8421, 2021.

[Kohl and Stone 2004] N. Kohl and P. Stone. Policy gradient reinforcement learning for fast quadrupedal locomotion. In *IEEE International Conference on Robotics and Automation, 2004 (Proceedings)*, volume 3, pages 2619–2624, 2004.

[Llopis-Albert *et al.* 2019] C. Llopis-Albert, F. Rubio, and F. Valero. A review of mobile robots: Concepts, methods, theoretical framework, and applications. In *International Journal of Advanced Robotic Systems*, volume 16.2, 2019.

[Mansour *et al.* 1999] Y. Mansour, D. McAllester, S. Singh, and R. S. Sutton. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, volume 12, 1999.

[Mosavi and Varkonyi 2017] A. Mosavi and A. Varkonyi. Learning in robotics. In *International Journal of Computer Applications*, volume 157.1, pages 8–11, 2017.

[Nasteski 2017] V. Nasteski. An overview of the supervised machine learning methods. In *Horizons. b*, pages 51–62, 2017.

[Perumal 2014] L. Perumal. Representing rotation in simulink using quaternion. In *Applied Mathematics & Information Sciences*, volume 8, pages 267–272, 2014.

[Ruder 2016] S. Ruder. An overview of gradient descent optimization algorithms. In *arXiv preprint arXiv:1609.04747*, 2016.

[Tsitsiklis and Konda 1999] J. Tsitsiklis and V. Konda. Actor-critic algorithms. In *Advances in neural information processing systems*, volume 12, 1999.

[van Otterlo and Wiering 2012] M. van Otterlo and M. Wiering. Reinforcement learning and markov decision processes. In *Reinforcement learning*, pages 3–42, 2012.

VIII. Appendix

The video recordings for select runs using the non-Baseline primitive and combined reward structures can be seen at this GitHub repository. Runs for each reward type showcase the typical locomotive behaviour produced by that reward, specifically with relation to their speed and stability.

Fig. 2. Returns during training for all reward types

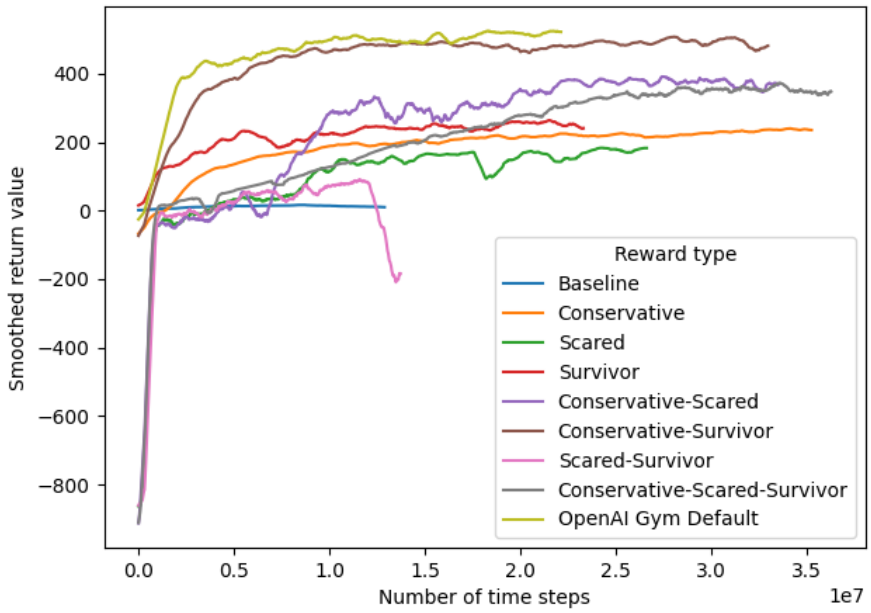


Fig. 3. Total loss during training for all reward types