RESEARCH REPORT - MaRBLe 2018/2019

Learning a locomotion controller for a quadrupedal robot with Deep Reinforcement Learning

Mathieu Coenegracht, Charlotte Dalenbrook, Kaspar Kallast, Samuel Kopp

SwarmLab, Department of Data Science and Knowledge Engineering

Supervisor: Dr. Rico Möckel

Maastricht University, Maastricht, The Netherlands

July 7th, 2019

1 Abstract

This paper explores how a deep reinforcement learning algorithm can be implemented in the context of a quadrupedal robot learning to walk. We analyze how different sensor and motor configurations affect the performance of the learning algorithm on three different robot models. We then test the same learning algorithm with the best performing robot setup on rough terrain. Additionally, we explore how changes in morphology affect the performance of a robot in simulation, and use our results to recommend a morphology according to the circumstance the robot is in.

Contents

2 Introduction [Samuel, Mathieu]

This paper was written as part of the MaRBLe 2.0 program. MaRBLe 2.0 is the research track of the honours programme of The Department of Data Science and Knowledge Engineering at Maastricht University. The paper focuses on the design of a locomotion controller for a quadruped robot through deep reinforcement learning. Legged robots have the ability to traverse complex terrains. Possible use cases include rescue operations after national disasters, data collection in dangerous environments and assisting in hospitals where they might have to traverse stairs. However, manual engineering of locomotion controllers is a highly complex control task that requires a very accurate dynamics model and a high level of expertise[2][3][17]. Recent developments in the field of deep reinforcement learning have produced impressive results in learning controllers[11][33][32][38] [12]. We originally set out to learn a robust control policy that is be able to adapt to different terrains and transfer it to a physical robot. We defined the following research questions:

1. Can we guide the learning process of adaptive rough terrain traversal effectively?

2. To what extent can hardware alterations such as the full rotation of the legs, the legs length, and the type of foot improve the performance of a quadrupedal robot over rough terrain?

Question 2 is explored in this paper. However, because of time limitations and several hurdles, such as needing to switch simulators, we were unable to sufficiently answer question 1. Instead we divide the paper in 3 subsections. The first section describes the process of learning locomotion control through deep reinforcement learning. Different observation and actions spaces are explored and their effect on learning a controller is studied. We describe our robot model and the different simulators that were used for our experiments. In section 2 we describe learning on a variety of terrains. Exploring different properties of terrains. Describing how terrains can be generated. In the last section we explore how changes in the robot's morphology affect its ability to traverse on flat and rough terrain.

3 Learning Locomotion Gaits

3.1 Introduction [Samuel]

Quadrupedal locomotion control is a high-dimensional continuous control task which can be formulated as a reinforcement learning problem. In recent years the combination of reinforcement learning and neural networks has shown great results not only in video games, board games and for scientific discovery, but also in the robotic control domain. [37] [24] [19] We employ the state-of-the-art deep reinforcement learning algorithm called Proximal Policy Optimization(PPO) to learn the control policy for a quadrupedal locomotion gait.[36] Even though

many experiments with DRL have been done on simulated and real applications, we find that not enough empirical evidence for consistent results on different robot model setups was shown. This is why we want to answer the research question of how the PPO algorithm performs with different choices of actions and observations under the case study of three robot models.

3.2 Related Work [Samuel]

The concept of training a neural network as a controller for locomotion was probably inspired by nature. Animals evolved with neural circuits that allow their limbs to move in rhythmic patterns while quickly processing the incoming sensor information for adapting their gait to not lose balance. Artificial Central Pattern Generators (CGGs) are a classic approach to reproduce these rhythmic motions in robots. [15] Already decades ago, researchers have trained CPG's as instances of recurrent neural networks with evolutionary algorithms and demonstrated adaptive robotic locomotion. The output of neurons does not have to be intrinsically oscillatory, but the network as a whole is capable of generating the necessary rhythmic control signals. [4] Later, reinforcement learning has shown significant improvement in training these network controllers. For example, Y.Nakamura et. al. designed a CPG-based actor-critic model for learning locomotion on a biped robot and demonstrated success in simulated environments and on a real platform.[25] [22] [28] Nowadays better learning algorithms and computing power that allows massive parallelization of simulation and neural network optimization, made deep reinforcement learning (DRL) part of the cutting-edge in robotic control research. For instance, the researchers from the Robotics System Lab at ETH Zurich successfully employed DRL to learn agile and dynamic motor skills such as recovering from a fall in the beginning of 2019.[18] Most recently a joint team of Google Brain, BAIR and UCB released a paper in which a sample-efficient DRL-algorithm based on maximum entropy was able to acquire a stable gait from scratch directly in the real world in just two hours without any simulation. [11] A promising family of DRL algorithms is based on trust regions of learning stability.[34] Based on this, J. Schulman et. al. developed the Proximal Policy Optimization algorithm, which still ranks well in many benchmarks of DRL domains. [21] This algorithm is subject of our work, because it is quite sample efficient for a policy gradient method and is relatively simple to implement and analyse. We believe that these properties make it ideal to be employed by engineers for robot control.

3.3 Methodology

3.3.1 Background [Mathieu]

We formulate the problem of locomotion as a Markov decision process (MDP). The MDP is a 4-tuple, (S, A, R, P), and consists of a state space S,an action space A and a reward function $r_t = R(s_t, a_t, s_{t+1})$ which gives the reward of taking action a_t in state s_t and ending up in state s_{t+1}. The cumulative reward $R = \Sigma_{t=0}^{\infty} \gamma^t r_t$ is called the return. $P(s' \mid s, a)$ is the transition probability

function, which gives the probability of transitioning to state s' when taking action a in state s. The lack of a perfect model of the robot and its dynamics make the process partially observable.

3.3.2 Proximal Policy Algorithm [Samuel, Mathieu]

The goal is to find a parameterized policy $\pi_\theta(a|s)$ that maximizes the expected return $J(\pi) = E_\pi[R]$. We sample actions from the from the policy and compute the log-likelihood $\pi_\theta(a \mid s)$ that action a was sampled from π_θ given state s. The policy is then optimized with gradient descent:

$$\theta_{k+1} = \theta_k + \alpha \Delta_\theta J(\pi_\theta) \mid_k \tag{1}$$

where:

$$\Delta_\theta J(\pi_\theta) = E[\Sigma_{t=0}^{T} \Delta_\theta log \pi_\theta(a_t \mid s_t) R] \tag{2}$$

In this implementation we use the advantage function $A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s)$ instead of the return, where $Q^\pi(s,a)$ is the on-policy action-value function and V^π is the on-policy value function. This can be estimated by:

$$A^{GAE(\gamma,\lambda)} = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+1}^V \tag{3}$$

where $\delta_{t+1}^V = r_t + \gamma V(s_{t+1}) - V(s_t)$ and can be considered as an estimate of the advantage of taking the action a_t with hyperparameters λ as a bias-variance trade-off and γ as a discount factor. This approximation method called Generalized Advantage Estimation results in faster and more stable policy learning. [35] This version of PPO uses a clip function in the objective to ensure that the updated policy does not get too far from the old policy. This makes a surrogate objective function:

$$L(s,a,\theta_k,\theta) = min[\frac{\pi_\theta(a|s)}{\pi_k\theta(a|s)} A^\pi(s,a), clip(\frac{\pi_\theta(a|s)}{\pi_k\theta(a|s)}, 1-\epsilon, 1+\epsilon) A^\pi(s,a)] \tag{4}$$

which can be updated via several stochastic gradient descent steps. Additionally the algorithm stops doing gradient steps when the KL-divergence of the updated policy from the old policy is above a threshold. This method was taken from the open-source implementation of PPO from the team of OpenAI.[30] The algorithm is built with an actor-critic model, where both the policy and the value function estimator are multi-layer feed-forward networks implemented in Tensorflow 1.13, consisting of 2 layers with 64 hyperbolic tangent units. Multiple actors gather experience in the form of reward, sampled action from the policy, state-observation and log state-action probabilities by acting in the simulated environment.

Algorithm 1 PPO, Actor-Critic Style

for iteration=$1, 2, \ldots$ **do**
  for actor=$1, 2, \ldots, N$ **do**
    Run policy $\pi_{\theta_{\text{old}}}$ in environment for T timesteps
    Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
  end for
  Optimize surrogate L wrt θ, with K epochs and minibatch size $M \leq NT$
  $\theta_{\text{old}} \leftarrow \theta$
end for

Figure 1: PPO as described in Proximal Policy Optimization Algorithms [36]

3.3.3 Observation Space [Mathieu]

When interacting in the real world an agent will often not have complete knowledge of its environment. Because of the high dimensionality of the environment's state space for locomotion control tasks, decisions have to be made on the choice of observation space. We are interested in the effect that the choice of observation space has on the agent's ability to successfully learn locomotion control in a partially observable environment. Common choices of the state vector in locomotion control through deep reinforcement learning are joint or link positions and velocities [27] [31] [32]. In addition the robot's body orientation and angular velocities are often included in the observation space[11]. Lillicrap et al managed to succesfully tackle a variety of challenging problems using pixels as observations.[20].Duan et al performed a similar study on the effect of a limited observation space on basic control tasks. [8] Since the required time increases with the size of the observation space [40], we are interested in finding an observation space that has a good trade-off between the time it takes to learn the policy and maximizing the return.

Five different observation spaces are compared, with the state input vectors ranging in dimensionality between 19 and 257 dimensions. Observation space 1 (OS1) consists of the position and orientation of the models torso and joints and has 19 dimensions. Observation space 2 (OS2) consists of the torso and joint positions, orientation, linear and angular velocities. It has 37 dimensions. Observation space 3 (OS3) contains the same observations as OS2, in addition it contains the external forces applied to the body. The state input vector has 97 dimensions. Observation space 4 (OS4) consists of the same observations as OS2, in addition it contains the inertia and velocities of the centers of mass, it has 197 dimensions. Observation space 5 (OS5) contains the observations of both OS3 and OS4, and has 257 dimensions.

3.3.4 Action Space and Motor Control [Samuel]

In this implementation of PPO the policy output consists a vector of means of normal distributions with a constant variance. A higher variance leads to more exploration in the action space with the trade off in stability. It is also possible to reserve a part of the policy output to determine variance or train a separate parameterized function for this purpose. Empirically this lead to a high increase of KL-divergence after each gradient step caused by the difference

of action likelihoods. We found that a constant variance of around 1/4 of the action range lead to best results and is thus used in all experiments.

Part of our analysis is how different choices of motor control and degrees of freedom impact the learning of a locomotion gait. Generally a setup that allows random actions to lead to rewards is necessary. An action space that directly exhibits the rhythmic motion of a locomotion gait should help in finding good actions early, but also constrains the space of possible gaits. Using a tuned PID controller can simplify the learning, but it has been shown that simple motor torques or velocities can already work.[33]
In our experiments we are using three different formulations of the action $\overrightarrow{a}$:

3.3.4.1 Direct Joint Angles

The actions are directly used as an input for joint angle target positions at each time step. The PID-controller for position control is inbuilt in Webots and computes the motor torques according to the following algorithm:

```
error = Pt - Pc;
error_integral += error * ts;
error_derivative = (previous_error - error) / ts;
Vc = P * error + D * error_derivative + I * error_integral ;
if (abs(Vc) > Vd)
  Vc = sign(Vc) * Vd;
if (A != -1) {
  a = (Vc - Vp) / ts;
  if (abs(a) > A)
    a = sign(a) * A;
  Vc = Vp + a * ts;
}
```

Figure 2: Screenshot of how the PID-controller recomputes the current velocity Vc [23]

3.3.4.2 Feedback controlled oscillation with two degrees of freedom(DoF)

The action $a_{i,j}$ serves as parameters for a phase-coupled oscillation of the target joint angle of motor j, such that:

- $a_{1,j} \leftarrow$ amplitude of front legs
- $a_{2,j} \leftarrow$ magnitude shift of front legs
- $a_{3,j} \leftarrow$ amplitude of back legs
- $a_{4,j} \leftarrow$ phase shift of back legs
- $a_{5,j} \leftarrow$ magnitude shift of back legs.

The phase is computed by $\phi(t) = f2\pi t$, where f is the frequency constant and the target position of the joint angles are computed by:

- $X_{front,j} = a_{1,j} sin(\phi(t)) + a_{2,j}$
- $X_{back,j} = a_{3,j} sin(\phi(t) + a_{4,j}) + a_{5,j}$

such that the range is limited to a given interval [-i,i] with $X = \tanh(X) * i$. If the actions are constant for each state, this would result in a galloping motion given the right choice of $a_{i,j}$. But being capable of adjusting the oscillation parameters given the current state allows the robot to react to external forces or different surface structures.

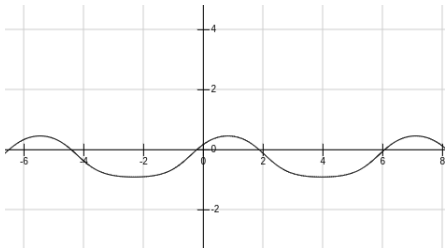


Figure 3: Oscillation with constant parameters X = $tanh(sin(t + 0.75) - 0.5)$

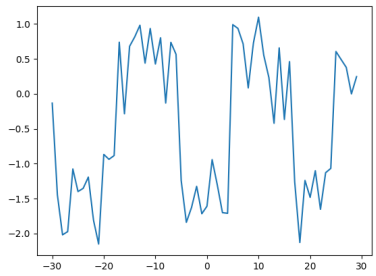


Figure 4: Trained feedback oscillator (actions sampled from policy)

3.3.4.3 Feedback controlled oscillation of shoulder joints

In this control mode only the shoulder joints follow the same pattern as in mode B). The other joints' angles are determined by the direct output of the policy. The output is here again constrained by the hyperbolic tangent to be in between the positions range.

3.3.5 Reward Function [Samuel]

We guided the learning to promising gait patterns by reward shaping. The reward function is parameterized by:

- Distance measure $d = c_1 * [x_{pos}(t+1) - x_{pos}(t)]/dt$ (x_{pos} means position on x-axis)
- Energy cost $e = c_2 * [battery(t+1) - battery(t)]/dt$
- Survival bonus $s = c_3 * dt$ if n legs have no contact with the ground ($n-1$ is the amount of legs the robot model can lift while staying in a stable position.)
- $r = d + s - e$

We empirically decided the parameters to be $c_1 = 150$, $c_2 = 0.00003$, $c_3 = 0.005$, which initially leads to rewards in the range of $[-1, 1]$ and $e - s \approx 0$ for a simple stable locomotion gait. The energy cost usually leads to more natural gaits, but can lead the algorithm into the local minima in which the best option is to find a terminating state fast to avoid a negative reward. This behaviour is avoided through the survival bonus, which on the other hand can lead to the policy seeking a state in which it simply rests in a stable position. This problem is solved by only rewarding survival if the robots legs are actually in motion and lifted.

3.3.6 Simulator [Kaspar]

Initially we chose to use the MuJoCo (Multi-Joint dynamics with Contact) simulator, developed by Roboti LLC, [39] to simulate a quadruped robot. MuJoCo was well suited to our needs- it had its own fast and accurate physics engine, it had a well designed format for designing models and it enabled users to modify the way the simulation is run (for example decide which of the simulator's processes are run and not run) which was handy for using learning algorithms in the simulator.

To test learning algorithms, we developed a crude and simple quadrupedal robot model in MuJoCo, it was based on the robot in the Swarm Lab. The robot had a torso with simple SLIP [9] legs. The robot model successfully learnt to walk after training it in the simulator.

Webots offers a selection of standard robotic sensors. These sensors can be attached to the robot model. A simulated battery is available for tracking energy usage. Webots also has motors, which fall into two types: rotational and linear. As the names suggest, rotational motors power rotational motion and linear motors provide linear motion. Motors in Webots are placed into 'device' fields of joint nodes. The degrees of motion freedom provided by the joint are actuated upon by their respective motors. Motors have various parameters used to define their behaviour such as maximum or minimum amount of motion allowed in a specified direction.

3.3.7 Robot Model [Kaspar]

3.3.7.1 Swarmlab Quadruped [Kaspar]

A three-dimensional model of the Swarm Lab's quadruped robot was made to be used in Webots. The model was made using the VRML97 (Virtual Reality

Modelling Language) used by Webots and additional features offered by Webots such as various sensors. The model is true to the general shape of the actual robot and has the same length measurements for all its components.

A robot's VRML file is structured as a hierarchy of nodes (parent-child relationship between nodes). A node is some entity such as a motor or some physical object like a cylinder. Nodes have fields which hold values (text or numbers for example), fields describe nodes.

To define the various physical parts of the model, Webots' solid nodes were used. Solid nodes represent an object which has physical properties such as mass and contact properties. Solid nodes also have bounding object fields which specify a geometrical primitive for collision detection. Solid nodes also need a shape node as a child to specify the solid's physical appearance. This shape node can also be used as the bounding object of the solid. Typically, if the shape is complex then using that very same shape as a bounding object is not a good idea, as it could cause complications with the collision detection calculations, so a sufficiently simpler geometry is used for the bounding object. In our case, the shapes used for the model were simple enough to allow us to use those same shapes also as bounding objects for their respective solids.

To allow movement for the model's limbs, joints were used. For the model's shoulders, Webots Hinge2Joints were used. A Hinge2Joint allows rotational motions around two intersecting axes (two degrees of freedom). For the model's knees, Webots SliderJoints were used, which allow a translation motion along a specified axis (one degree of freedom). Webots SliderJoints were also used to model the quadruped's feet with springs. These various joints allow the specification of spring and damping constants, these parameters were used to make the link between the feet and knees behave as a spring.

Webots offers motors in the form of rotational motors and linear motors. They were used to actuate the shoulder and knee joints respectively. These motors power their respective joints and thus produce motions along the joint's axes.

All physical components of the model use solid nodes. The model's main body, the torso, is at the top of the node hierarchy, all other components of the model are its children. For more details on the model's physical components, see the appendix section "Webots Model Tables" (the parts appear in the table in the same order as they are in the the model's node hierarchy). Details regarding the centre of mass of the model's components has been left out of the table. These values have been left at their default values given by Webots. Information regarding motors can be seen in the appendix, in the section "Webots Model Tables".

Neither the masses of the model's components nor the motor's values match up with the true robot's parameters for now, as emphasis has been on having the model work properly within the simulator. If however, for example a learnt control policy were to transferred from the model to the real robot, then one should consider matching the parameters of the model to the actual robot. The foot spring's spring constant value is 3000 (spring force in Webots is calculated according to Hooke's law: F = -Kx, where K is the spring constant and x is

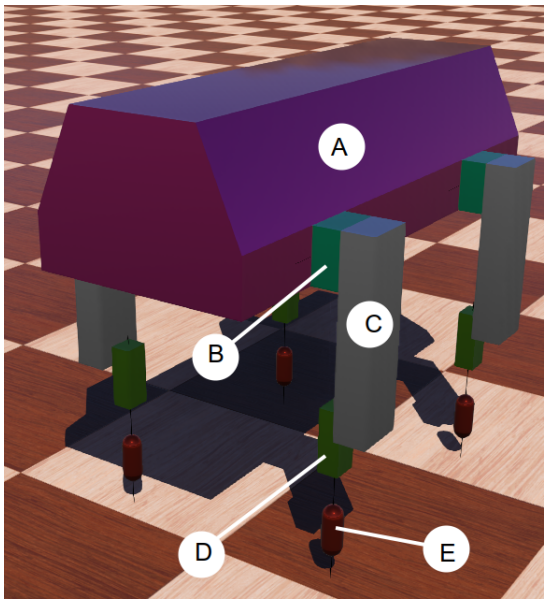


Figure 5: The quadrupedal model designed in Webots. Its parts are labelled by letters in the order in which the parts appear in the node hierarchy. A: Torso, B-D: Leg, E: Foot. The foot (part E) is connected to the leg by a spring, which is represented by slim black lines.

the current joint position). Again, this value does not match with the actual robot's, but for simulation purposes it works as required.

The total mass of one of the model's legs is 0.45kg and the total mass of the model is 14.8kg. The torso is far heavier than the legs and is somewhat higher up than the legs; this could potentially cause some issues with the model's standing stability. So far the model has not exhibited any such problems though. These values are subject to change within our applications of this model.

The original robot's legs can collide as it walks, the same can happen to the Webots model, however for now collision between the different parts of the model has been switched off to simplify the learning process and also having self collision activated can cause some issues with the touch sensor's bounding object (if the touch sensor is used). Looking at figure 4, part D can move up and down the slider joint that connects it to part C (the slider joint itself is invisible); from its original starting position it can move upwards by 0.085m and downwards by 0.01m. The entire leg is connected to the torso (part A) by the earlier mentioned Hinge2Joint. The entire legs is free to rotate around one revolution with the current settings, but this limit can be increased. The entire leg can also do abduction and adduction motions from and towards the torso; 0.5 and -0.5 radians for both abduction and adduction.

3.3.7.2 MuJoCo Dummy Quadruped [Kaspar]

For the MuJoCo simulator, a simple quadrupedal robot model with SLIP (spring load inverted pendulum) legs was designed [9]. The model was specified in MuJoCo's modelling format- an XML file in the MJCF format. A cuboid was used as the model's main body. Each of the four legs were made up of two cylinders connected by a spring. The model's intent was to be used as a quick and easy way to test out algorithms on. The model had the option to add various sensors to it as required.

3.3.7.3 Webots GhostDog [Kaspar]

The Webots simulator offers a choice of model robots. We used the quadrupedal "GhostDog" by the EPFL BioRob laboratory which resembles a dog in its qualities (we used the headless version). The model has passive knee joints (with dampers and springs) and active hip joints. This robot has fewer degrees of freedom than our MuJoCo's simple model and our Webots quadruped.

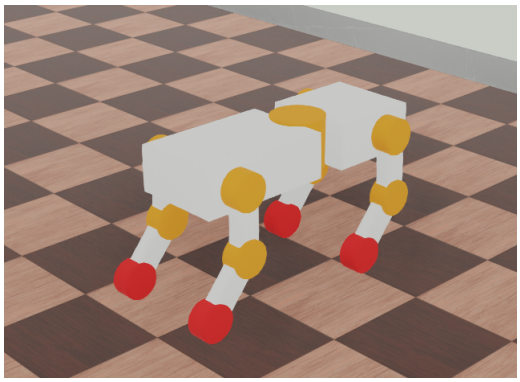


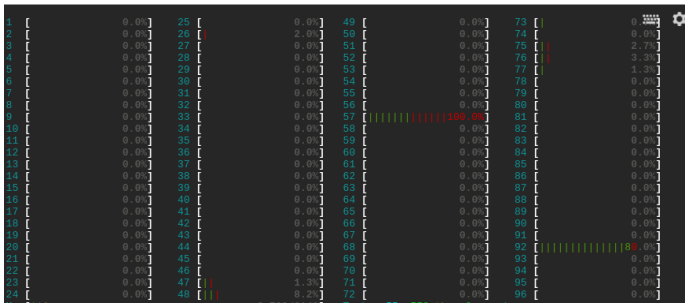
Figure 6: Headless GhostDog

3.3.8 Parallelization [Samuel]

When debugging and tweaking DRL algorithms the ability to run a lot of experiments in short time is crucial. On a 2.3GHz Intel i5 dual core machine with 8GB RAM a single robot-environment interaction in MuJoCo takes around 0.14 seconds including the reset of the simulation after each trajectory. For PPO as an on-policy algorithm this is the major bottleneck of training speed. If enough cores are available this process can be distributed to several workers gathering experience in parallel. To enable this we use a Google Cloud Instance with 96 CPUs and 624 GBs of RAM. Because MuJoCo requires a commercial license to run on Virtual Machines, only Webots experiments are run on this.

A central python script starts multiple Webots instances at the beginning of each epoch and receives the gathered experience data as binary files on memory.

Even though the instances run in parallel, due to non-determinism in Webots they do not finish at the same time.



(a)

(b)

Figure 7: (a) cloud instance waiting for last processes to finish (b) cloud instance cpus in full use

This makes the script wait for some instances before starting the central processing of the data and running gradient steps of the neural network. In practice the instance uses on average 70 percent its processing resources. With this setup a simulation step takes around 0.09 seconds. We concluded that the speedup of using Webots on the cloud instance is not significant in comparison to using MuJoCo on a local machine. The main reason for this is that Webots has to restart the control script every time the simulation is reset. This way the Tensorflow graph that contains the current control policy has to be reloaded every time from scratch.

As an alternative to file communication the main script can function as a local server and communicate with Webots via sockets. But in the tests we did this did not lead to major speed improvement, since it does not solve the reloading issue. An option that was not tested, was serving the Tensorflow output directly from one graph to each Webots instance instead of storing and loading the graph each time. This might speed up the reloading time significantly, but it is not clear to us how this can be implemented.

3.4 Experiment Setup [Samuel, Mathieu]

The PPO hyperparameters are picked from the implementation from the OpenAI SpinningUp repository.[**?**] In all cases: $\lambda = 0.97$ for GAE-Lambda, discount factor $\gamma = 0.99$, policy optimizer learning rate is 0.0001, value function optimizer learning rate is 0.001, ratio for clipping the policy objective is 0.02 and a KL-stop of 0.01.

3.4.0.1 Observation Space

All experiments were conducted in MuJoCo using the quadruped model described in section 3.3.7.2. For each observation space the learning algorithm was ran 10 times with different random seeds. Each trial run consists of 500 epochs, each containing 4000 simulation steps with a maximum trajectory length of 1000 steps. For each epoch we compute the average return over all trajectories. We then compute the mean return and standard deviation over the 10 runs for each epoch.

3.4.0.2 Motor control [Samuel]

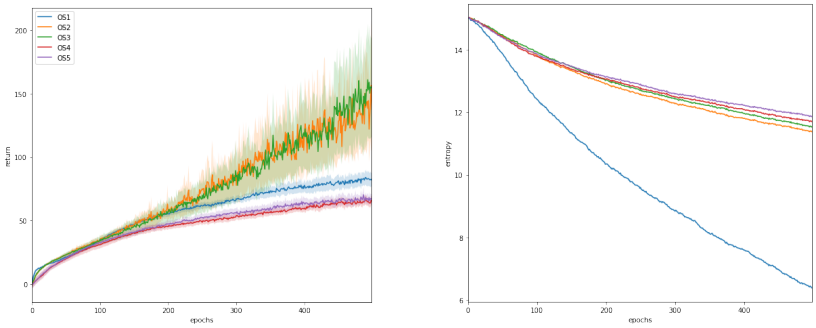
All motor control experiments are done in Webots using the Swarmlab Quadruped model and the Ghostdog model. Each experiment is run with 3 different random seeds. The experiments were each run for 50/100 epochs with 1000/2000 steps per epoch and a step limit of 500 per trajectory for the Swarmlab Quadruped/GhostDog. Because of time constraints we took less time steps for Model A. An epoch takes around three time as long for the Model A, because the time before the robot falls is much shorter which leads to more simulation restarts.

3.5 Results and Discussion [Samuel, Mathieu]

3.5.0.1 Observation space

The solid line in figure 6a shows the average return over 10 different seeds. The shaded region is half the standard deviation. Figure 6b shows the approximate entropy averaged over the 10 runs. The algorithm performed better when using OS2 and OS3 compared to the other observation spaces. OS2 and OS3 have comparable performance. The algorithm achieved slightly better performance using OS1 compared to the use of OS4 and OS5. The algorithm has near identical performance between using OS4 and OS5. The approximate entropy using OS1 is much lower compared to the other observation spaces. The low entropy of OS1 indicates little exploration compared to the other observation spaces.

A possible explanation for the poor performance when using OS1 could be lack of ability to predict the velocities from only the robots position. Duan et al. found that recurrent networks outperformed feed-forward networks on basic control tasks when the input consisted of only positional observations[8]. Adding the external forces on the body in the observation space did not seem



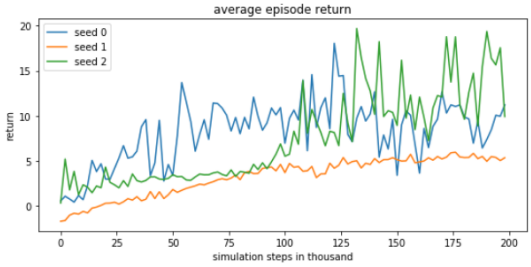
(a) return (b) entropy

Figure 8: (a) Average return with half the standard deviation. (b) Approximate entropy

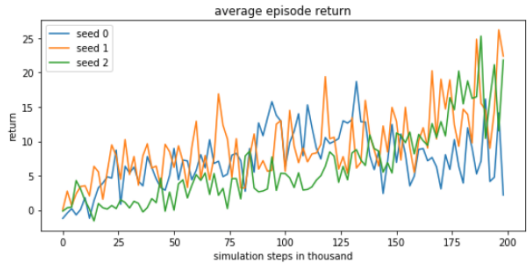
to significantly increase the performance. This could perhaps be attributed to the large increase in the size of the observation space. The size of OS3's state vector is more than twice that of OS2. However, this does not explain the similar performance between OS4 and OS5. The size of OS5 is larger than that of OS4, yet the performance of the algorithm is nearly identical between the two. A possible explanation is that adding the velocities of the centers of mass to the observation space does not provide much information gain over having only joint velocities, while the increase in the size of the observation space slows down the rate at which the algorithm learns. Having a wider variety of information could possibly explain why when using OS5, despite its larger size, the algorithm achieves similar performance.

3.5.0.2 Action Space and Motor control [Samuel]

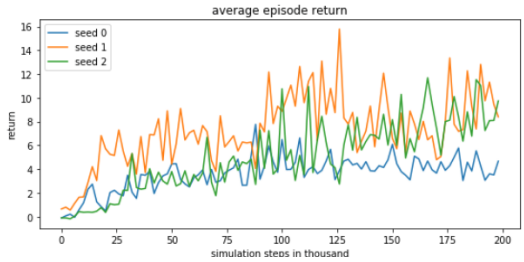
a) Swarmlab Quadruped: None of the different choices of motor control lead to success in learning a locomotion gait. In each case the return only initially increased and then stayed approximately the same for the rest of the learning. After visualizing the training it was clear that the policy could not achieve better behaviour than falling in the right direction for a small reward. A reason could be simply that this robot model is not capable of keeping its balance, no matter how optimized the controller is. The contraction joints appeared to behave chaotically in most trajectories possibly due to unrealistic spring parameters. Also the torso shape and leg-body ratio might lead to immediate falls when a leg is raised. Another possibility is that random actions lead on this robot model rarely to do a balanced step in the right direction and if they do then a single policy gradient step is not sufficient to leverage this behaviour consistently. An off-policy learning algorithm with an experience buffer might suffer less from this problem.



(a)



(b)



(c)

Figure 9: (a) Direct Position Control (b) Feedback oscillation (2DoF) (c) Feedback oscillation (6DoF)

b) Ghostdog model: The only consistently stable gait was learned by the the feedback oscillation method with two DoFs. This is probably simply because given that parallel joints are actuated with the same input, the robot automatically balances. Then the learning task is reduced to avoiding falling over the front or back and increasing the speed. With both the direct position control and the feedback oscillation with six DoFs, a stable gait could not be archived. In both cases it manages to run enough steps to almost get similar rewards in comparison to the version with two DoFs. But then it is not walking consistently into the right direction and cannot stay in the gait without falling. In these cases the desired monotonic improvement of PPO could not be achieved and looking at the variance of the returns between the epochs the algorithm appears to be very unstable. An important note is that even when comparing the policy learned from method B, a simple set of tuned constant parameters

for the oscillation does perform significantly better. The advantage of reacting to changes in state perturbing the harmonic motion gets lost of course. Many papers presenting the effectiveness of their reinforcement learning algorithm may just achieve this with the optimal hyperparameters, choice of variance, reward function or simply trying out different setups. Perhaps this is generally a necessary work that comes with the trade-off of less domain knowledge. We cannot be sure if our results would have not been different if we just searched for the right parameters or increased the training time.

3.6 Conclusion and Further Work [Samuel, Mathieu]

Including joint velocities significantly increases the algorithm's ability to learn a policy versus having an observation space consisting solely of positional information. Including external forces applied to the body did not seem to provide a significant increase in performance over using joint positions and velocities. In future work we could test if this can be attributed to the increase in training time caused by the larger size of the observation space. The inclusion of inertia and velocities of the centers of mass caused a large drop in the algorithms performance. However, when using OS5, that in addition contains the external forces and has a larger state vector, the performance was nearly identical. We hypothesize that this can be attributed to the velocities of the center of mass and rotational velocities being similar observations, and that the large increase in dimensionality and small information gain leads to large drop in performance. In future work we could test this hypothesis by comparing several observation spaces that are identical except for that one contains the joint velocities and the other contains the velocities of the centers of mass.

We conclude that the choice of motor control, especially the degrees of freedom makes a significant difference in our test cases. The PPO algorithm seems to lack exploratory power and has troubles finding with stable learning when the dimensionality is higher.

Success of the algorithm in one robot domain does not necessarily guarantee that it is capable of learning a gait on another robot model or with a different motor and sensor setup. Nevertheless we want to note that more experiments are necessary to determine if perhaps different hyperparameters or longer training times can change this conclusion.

4 Learning on Different Terrains

This section describes an approach to reinforcement learning with a terrain generator for learning locomotion on varying courses using the Webots simulator.

4.1 Introduction [Kaspar]

Rough terrains are prevalent, hence it would be good to capitalize on the quadrupedal robot's potential capability of crossing such terrains. Doing so would make legged robots much more versatile and competent when in locations with difficult terrains. Robots with the ability to cross rough terrains would be useful in many applications in the world, such as: search and rescue missions, industrial inspections at gas and oil sites [5] and humanitarian demining [7].

A script for generating various terrains with different features for the Webots simulator is described. The purpose of these terrains is to be used for learning and experimenting locomotion for a quadrupedal robot model. Our learning algorithm is tested on some of these terrains.

4.2 Related Work [Kaspar]

For traversing varying terrains and environments, reinforcement learning has been used to achieve successful locomotion. Deepmind made use of various approaches to reinforcement learning [12]. A planar walker, humanoid and quadruped were used for learning good locomotion skills.

Deepmind also made a procedural terrain generator for the agents learning locomotion in the MuJoCo simulator [12]. Namely, in each training episode a new course is generated. The course can have a variety of features, such as: gaps in the ground, ramps, hills, platforms and hurdles.

Reinforcement learning was also used for learning terrain adaptive locomotion capabilities [31]. In a 2D environment, a quadrupedal and bipedal agent were trained. A mixture of actor-critic experts method was used. High-dimensional terrain and state descriptions were taken as inputs, parameterized jumps and steps were the outputs.

Like Deepmind, also various classes of terrain obstacles were used [31]. These include: slopes, gaps and steps. The terrains were generated randomly with these features, drawing from ranges of values for the parameters that describe a kind of obstacle.

4.3 Methodology

4.3.1 Simulator [Kaspar]

To apply learning algorithms on the robot model, we needed to be able to run several instances on virtual machines, which would require multiple MuJoCo licenses, which we could not get. Thus we opted to use the Webots simulator instead (which has a free license). Webots is a simulation environment made by Cyberbotics Ltd for simulating mobile robots [23].

We used Webots to develop a more accurate model of the quadrupedal robot in the Swarm Lab. Like MuJoCo, Webots too was well suited to our simulation needs, but with its strengths and weaknesses in different areas. For example, Webots always needs the simulation application window to be open for every single simulation being run at the same time, whereas MuJoCo does not have this issue. On the other hand, Webots has a much better user interface, allowing the user to edit various scripts and models whilst the simulator is running.

4.3.2 Terrain Properties/Generation [Kaspar]

4.3.2.1 Overview [Kaspar]

A terrain generator was designed to facilitate learning the traversal of various kinds of terrains for the quadrupedal model in Webots. The user is able to specify the desired features of the terrain in the Webots supervisor script. Once the Webots world file is run, a terrain corresponding to the parameters is generated.

4.3.2.2 Controller and Supervisor [Kaspar]

In Webots, a controller script is linked to a robot. This controller script can be used to interact with the robot (e.g. get sensor readings or send motor commands). However, this controller script cannot be used to interact with other things present in the current Webots world. For this, a supervisor script is required. Supervisor scripts offer the same functionality as controllers, but also other functionalities which enable you to interact with all other things present in the Webots world. A supervisor script enables you to get handles to nodes and to their corresponding fields which describe the nodes. These handles can be used to change the field values. For example, the supervisor accesses a cube's node. It then from the cube's node gets access to its size field and then changes it as desired.

4.3.2.3 Main Platform [Kaspar]

The simplest terrain that can be generated is a flat platform (15*15m, but can be changed as needed). Using the platform's properties, a starting and goal position is calculated for the robot. Initially, a new robot was supposed to be spawned in using the supervisor, however, due to a bug in the Webots' supervisor whenever the robot model was spawned in, it had a wrong centre of mass, instead of being in the original place, the centre of mass had shifted forwards in the robot, causing it to fall down. So instead of spawning a new robot into the world, a robot is defined before the world file is run, and at run time the already defined robot's translation field is updated to the previously calculated starting position. The starting and ending positions are at opposite ends of the platform, each of them 1.25 meters from the ends of the platform to give the robot some space. Care needs to be taken to make sure that the platform is thick enough, so that the robot model does not fall through the platform- a thickness of 1 meter is sufficient. If desired, the platform can be slanted at a set angle or at a randomized angle. The platform is used as the default ground for the robot instead of the actual Webots world floor because it is easier to make changes to the platform.

4.3.2.4 Obstacles [Kaspar]

The terrain generator allows spawning either a random or desired number of obstacles onto the platform. Unlike the robot, spawning in new obstacle objects into the world does not cause the bug where the spawned entity's centre of mass is changed. The obstacles can have randomized sizes for diversity (and hence a new resulting mass too). The default obstacles are cubes, with lengths of 0.1m and a mass of 0.8kg. For additional diversity, the shapes in which the obstacles appear can also be randomized.

4.3.2.5 Elevation Grid [Kaspar]

For a more challenging terrain, an elevation grid can be created above the main platform (like the main platform, it too can be slanted). In Webots, an elevation grid is most suitable for modelling uneven terrain. An elevation grid is a uniform rectangular grid with differing heights in the y=0 plane of the local coordinate system. It has the following properties: xDimension, zDimension, xSpacing, zSpacing and an array of height values. X- and zDimension values determine the number of values in the height array in the x and z directions. X- and zSpacing fields determine the amount of space between each height point.

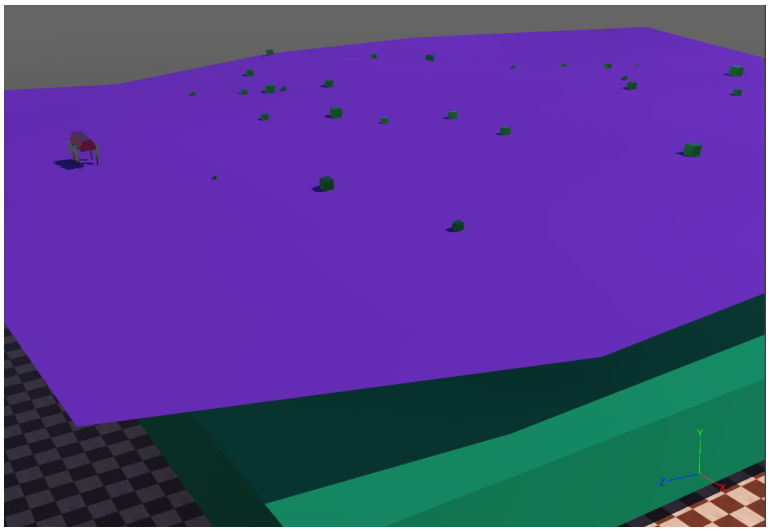


Figure 10: A sample terrain with a very smooth elevation grid over the main platform with some obstacles

A predefined elevation grid is spawned above the main platform. The default elevation grid can be used, but randomizing the elevation grid offers more variety. All of the above mentioned parameters can be randomized, but it is sufficient to only randomize the height values in the height array and the x- and zSpacing values, whilst the x- and zDimension values both are left as 5. The elevation grid's smoothness depends on the randomized parameters. To ensure that the terrain remains relatively smooth and feasible for the robot model, the Gaussian distribution is used to determine the height array values with a mean of 0.575m and a standard deviation of 0.06m. The Gaussian distribution is also used for both the x- and zSpacing values, with a mean of 1.1m and a standard deviation of 0.2. These values were obtained by testing out various values and seeing the resulting elevation grid. As with the main platform, it is necessary to make sure that the thickness of the elevation grid's base is thick enough to eliminate any risk of the robot model falling through it.

To avoid the risk of the newly randomized elevation grid being too small compared to the main platform, the elevation grid is fit to the platform's size. This is done by calculating a scale value from the spacing and dimension values of the elevation grid, and the lengths of the main platform. This obtained scale values is then applied to the elevation grid. Doing so ensures that either the x or z length of the elevation grid is exactly as long as the respective length of the main platform and the other elevation grid length value is greater than or equal to the main platform's respective length (it is not an issue if either of the elevation grid's lengths is greater than the main platform). It would have been better to have both elevation grid lengths exactly equal to the respective main platform lengths, by applying two scale values instead of one to the elevation grid, however Webots does not allow the applied scale values to be different from each other.

A bug was encountered when the parameters of the predefined elevation grid were altered. Once the height and spacing values were updated, the visual appearance of the elevation grid updated correctly, however the actual bounding object (for collision detection) remained unchanged. To get around this issue, updating the translation field of the elevation grid after updating its parameters suffices.

When the main platform is used, it is simple to position the robot into the calculated starting position as the height of the platform does not change. With the elevation grid this is not the case, as the height always changes, so one single starting height does not suit all occasions, as the robot can spawn directly inside the elevation grid. One solution is to always drop the robot from a certain height onto the elevation grid. However, the elevation grid can be uneven at the drop site so the robot could immediately fall over upon contact with the elevation grid.

4.3.2.6 Stabilization and Physics Plugin [Kaspar]

One solution to this issue is to apply a stabilizing force to the robot for a short while (10 seconds) to help the robot stay upright. This requires a separate physics plugin to be made in Webots. The physics plugin is used to apply forces to a selected object, in this case the robot. The simplest way to do so is to apply a constant force of 135 newtons to the robot upwards in the y-axis. This crude method is fairly successful, as can be seen in the experiments section. For more accurate forces, inertial unit readings of the robot can be sent from the robot's controller to the physics plugin. The sum of a constant force (e.g. 60N) and a scaled combination of the inertial unit readings is then applied to the robot upwards in the y-axis. If the force is too big, then there is a risk of the robot flying upwards. To counter this, whenever the force exceeds a certain limit, it is set equal to the limit (the limit is 160N, and around 200N makes the robot fly).

Another way to help the robot stay stable upon landing on the elevation grid is to reset the robot's physics upon contact with the elevation grid (a touch sensor attached to the robot is used to detect collision between the robot and

the elevation grid). Doing so sets the linear and angular velocities of the robot to zero and hence also the inertia is set to zero. This method and applying stabilizing forces to the robot can be used simultaneously for more efficiency.

Instead of dropping the model, it could also be possible to simply have the model begin directly on a flat platform adjacent to the elevation grid, with the intent of the model then proceeding to the elevation grid. However, if the elevation grid has steep differences in heights, especially at where the starting position platform and the elevation grid connect, then it would be a severe challenge, if not impossible for the model to even at a fully trained state to mount the elevation grid from the platform. If it is important that the robot is to be on only an elevation grid throughout the learning experience, then this starting platform method would not be suitable.

4.3.2.7 Contact Properties [Kaspar]

It is also possible to change contact properties between the robot and the objects present in the terrain generator. In Webots, contact properties have many parameters which can be changed, however for now the terrain generator is limited to changing the Coulomb friction and the coefficient of restitution parameters of the contact properties for simplicity. Modifying contact properties can be useful for replicating various kinds of terrains with different frictions and restitution coefficients. The change to the contact properties can be manually set or randomized.

4.4 Experiment Setup [Kaspar]

All the experiments were done in the Webots simulator. Ideally, the learning and terrain generation processes would have been merged, however due to time limitations this was not done. Instead of having the terrain generator generate a new terrain for the learning process automatically as required, a terrain was generated which was then manually set as the learning environment's terrain. As a result, each training session relied on a single terrain, rather than getting randomized variants of the terrain when needed.

The model modelled after the Swarm Lab quadrupedal would have been used for these experiments, however because of difficulties with it achieving successful locomotion, the Webots GhostDog model was used instead. It had two degrees of freedom.

The first experiment was done on a very smooth and flat elevation grid. Three trials were run, within each trial 100 epochs were done, where each epoch had 2000 episodes.

The second experiment was done on a more difficult elevation grid- it had far greater height differences and much steeper slopes. The same three trials were run as before. In the three trials, the robot's starting position was changed per trial: in trial 1 the robot started at a flat low point, in trial 2 another different flat low point and in the 3rd trial the robot began from the peak of a hill.

4.5 Results [Kaspar]

Figures 9 and 10 refer to experiments on the smooth elevation grid. Figures 11, 12 and 13 refer to experiments on the harder elevation grid. The seed values refer to a trial, (seed 0 = trial 1, seed 1 = trial 2 and seed 2 = trial 3). Additional results are in the appendix.

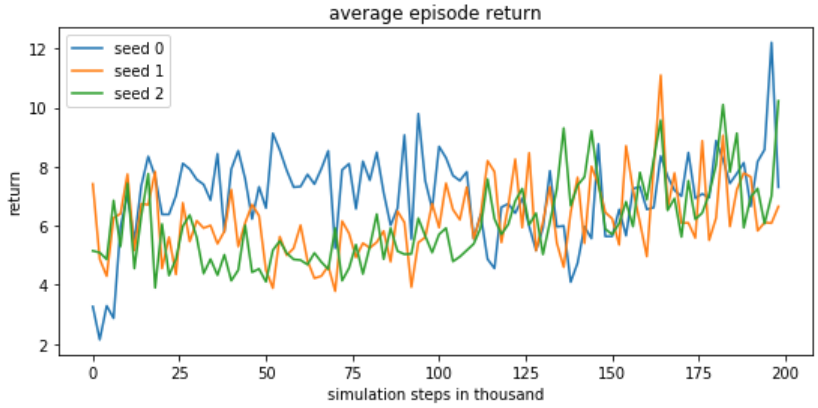


Figure 11: Easy elevation grid average return

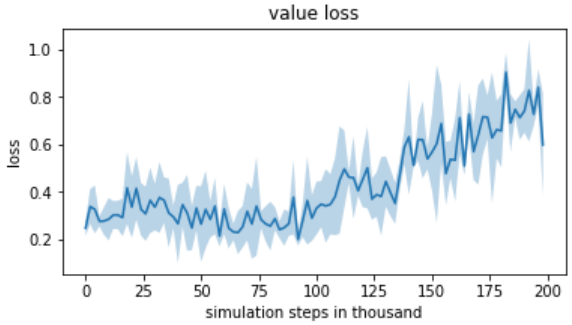


Figure 12: Easy elevation grid value loss (trials 1, 2 and 3 combined)

For the three easy and smooth elevation grid trials, average episode return increased slightly over time, with heavy fluctuations in all three trials.Trials two and three behaved quite similarly, whilst trial one was a bit less stable- it was somewhat higher than trials one and two for the first half. In the second half trial one behaved like the other two trials, except for one large peak in the very end. Value loss did not change much for the first half, however in the second half a significant increase occurs, going from 0.3 to 0.8.

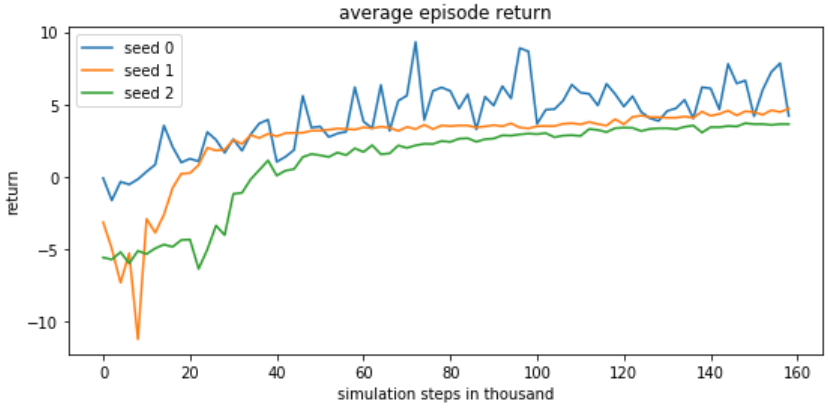


Figure 13: Hard elevation grid average return

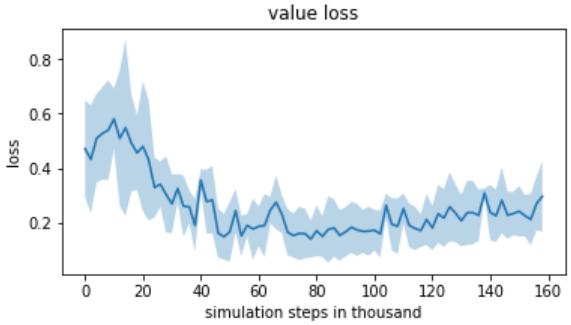


Figure 14: Hard elevation grid value loss (trials 1, 2 and 3 combined)

For the hard elevation grid, in figure 11, the three average episode returns started off at different values. After 40000 simulation steps, trials 2 and 3 were very similar, where they both increased a very small amount for the rest of the simulation steps. Trial 1 also ceased to increase significantly beyond 40000 simulation steps, but was much less steady than trials 1 and 2. Value loss (figure 12) decreased somewhat steadily until 40000 simulation steps, after which it stopped decreasing and fluctuated at around 0.2. Average episode lengths (figure 13) for trial 2 and 3 were again similar, both remained very low at around 50 simulation steps after some peaks in the beginning. However, trial 1 had short episode lengths in the beginning, then increased to around 150 steps. Trial 1 then fluctuated around 150 steps and slightly decreased until the end.

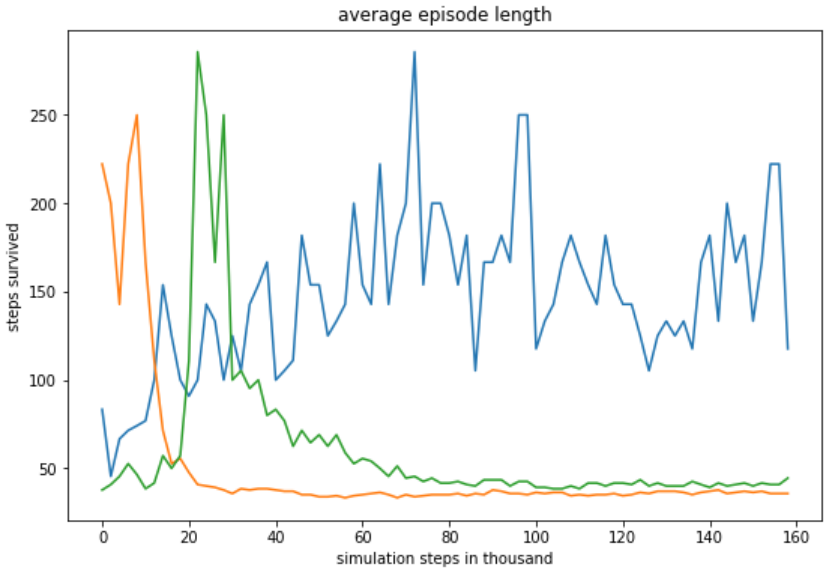


Figure 15: Hard elevation grid average episode length

4.6 Discussion [Kaspar]

In all three trials on the easy and smooth elevation grid, the robot model learnt to sustain a successful gait. In trials two and three, the model acquired a gait in which it would drag itself forwards with its two front legs whilst minimally moving the two rear legs. Such a gait works, however it does not promise much in terms of agility and speed, as only using the font legs to move is not using up all of the robots potential- it is more desirable that the robot moves using all four legs.

The value loss for the hard elevation grid trials decreased until 40000 steps, after which it became stable. The average return matches this behaviour, as it too increases until around 40000 steps after which it stabilizes. So at that point the policy is no longer exploring new states. An increase was expected for the value loss as we expected it to continue exploring new states, but instead it decreased and remained low.

Out of the three trials for the hard elevation grid, only the first trial managed to make the robot learn locomotion. In the other two following trials, the robot started to behave in a similar pattern where it would raise a limb and right after that it would fall. This is shown in figure 13, where the successful first trial has long episode lengths where it moves successfully and the other two trials have very short episode lengths due to the robot always almost instantly falling down and causing the episode to end.

It is not surprising that the third trial failed to learn proper locomotion as it started on top of a peak on the elevation grid, a challenging starting position.

However, trial two began on a lower and flatter area, yet it still failed to learn locomotion, unlike trial one.

It should be noted, that we did not test if the learnt gait on rough terrains is suitable in general for many rough terrains, or if it over-fits to the specific terrain it was trained on. Also, we do not yet have a way of determining whether a successful gait that the robot learns on a rough terrain is successful because the gait is inherently stable, or because it actually learnt a gait that is especially stable on rough terrains.

4.7 Conclusion and Further Work [Kaspar]

The basic functionalities of the terrain generator were successfully implemented. Some additional functionalities such as the stabilizer for when the robot in the beginning of an episode falls on the elevation grid were also added. Simple experiments with a robot learning locomotion on elevation grids were done; the model's performance on an easy and relatively flat elevation grid was good, but its performance on a harder terrain was quite lacking, failing to learn locomotion two out of three times.

For better experiments and performance of the robot in learning, the learning procedure and the terrain generation should be connected, such that at each learning episode, a new terrain is presented to the robot. Some form of heuristic for the terrain generation should be implemented such that the terrain's properties and difficulty change according to the robot's performance, to facilitate learning for it.

The issue of whether the model's learnt gait over-fits to one specific terrain or not should be tackled, if learning locomotion experiments are to be continued for the model on rough terrains. The same applies for whether the robot's gait is inherently stable or if it specifically works well for a rough terrain.

Using reinforcement learning to learn locomotion on rough courses in the Webots simulator shows promise. For our implementation to be more successful, some improvements should be made, but the general framework is now in place for doing so.

5 The Effect of Morphological Changes on the Performance of a Quadrupedal Robot on Different Terrains [Charlotte]

5.1 Introduction

Although rough terrain traversal has been the focus in a lot of robotics research in past years, not much has been done in comparing different morphologies to one another and how they perform on different terrains. By analyzing the effect that different morphologies have on the performance of a robot we want to reach

conclusions about which morphology is best to apply in different situations. We attempt to answer the following research question:

To what extent do hardware alterations such as the full length of the legs, the leg type, and the number of degrees of freedom available change the performance of a quadrupedal robot over both rough and flat terrain?

With answering this question we want to generate a recommendation to others for which morphology of a quadruped to employ according to the terrain type they are trying to cross. Recommendations we make include which leg length to use, which type of leg to use, and the number of degrees of freedom to give the legs.

5.2 Related Work

When designing a quadruped robot, emphasis is often placed on the energy consumption of the robot and its ability to conquer rough terrain (e.g. [13], [6]). Some of the most notable work for quadrupedal rough terrain traversal was done at ETH Zürich with ANYmal [13]. ANYmal implements modular joints that can be fully rotated, giving it a large range of motion and hereby enabling it to effortlessly walk and trot, even in irregular environments, as well as climb stairs [13]. Reasons for choosing certain parts of the robot include keeping the weight low, quick exchange of parts, the ability for application in many scenarios [13]. The Cheetah 3 quadruped robot was designed with a focus on keeping the cost of transport low [6]. This paper also includes an entire section on the design of the robot, with special focus placed on the legs, which include an extra degree of freedom, allowing it the legs to ab/adduct, unlike its predecessor [6].

Aoyama et al researched the optimal structure for a quadruped robot, with a focus on minimizing the joint torque sum [1]. They argue that finding the ideal limb length ratio for a robot is a vital part of enabling efficient terrain traversal for the environment that the robot is working in [1]. They found that joint torque increases if the slope angle the robot is walking on increases, and if the rear legs of the robot are shorter than its forelegs (and vice versa) [1]. Other papers focus less on the legs of the robot but instead explore how an active versus a fixed spine affected the gait of a robot [16]. In their research they concluded that the robot with active spine was both faster and more stable when going straight forward [16]. With DyRET, initial steps were taken to study how changes in the morphology of a robot affected its locomotion on rough terrain [29]. Each of the legs in the DyRET robot has 5 degrees of freedom [29]. The length of the legs can be changed dynamically from 25mm to 100mm [29]. This ability allowed the robot to keep performance high, even with varying servo torques [29].

Although a lot of research has been done on certain aspects of a robots' morphology, there is no clear definition of which morphology is good to use in certain scenarios. Although certain properties such as the modular joints in the ANYmal robot [13], or the high degrees of freedom of the Cheetah 3 [6] have shown to be useful in certain scenarios, there are no clear guidelines as to which

properties are useful when. One of our goals with this paper is therefore to build on the work of all of the aforementioned projects and generate a suggestion for the morphology of a robot according to a given scenario. In order to do so we conducted experiments using dummy robots in simulation and compared their performance. This allows us to generate suggestions to others who want to design a quadrupedal robot based on its working environment.

5.3 Methodology

In order to test the performances of different morphologies, several experiments were conducted. The experiments were all performed within a MuJoCo simulation and using the proximal policy algorithm described in section 4.4.2. Each experiment was conducted 5 times using a bash script which allowed for the experiments to be run consecutively. The three final return values after each trial of 500 epochs were written to a text file in order to be used for analysis.

5.3.1 Degrees of Freedom

The first aspect of the robot morphology that was tested was the number of degrees of freedom the legs of the robot had. For the degree of freedom experiments, we used a simulated version of the real-life robot we had. This robot has straight SLIP model legs with a slider joint for leg protraction and retraction, a hip joint that allows forward and backward movement as well as abduction and adduction. Additionally the bottom part of each leg consists of a spring structure that can be compressed slightly. These properties combined give each of robot's legs 3 degrees of freedom as shown in figure 2. In order to test how the presence of one of these degrees of freedom affected the robots' performance, we removed it and compared the average, maximum and minimum return of 5 trials over 500 epochs without that degree of freedom, to the average return when all 3 degrees are available. These experiments were conducted on both flat terrain and rough terrain to see if the results varied between the two.

5.3.2 Leg Lengths

The next aspect of the robots' morphology that was tested was the length of the legs of the SLIP model robot. Leg lengths were tested in increments of 0.1 from a length of 0.1 to 0.6. MuJoCo does not assume units therefore these lengths are relative to all other lengths of the robot. It therefore can be assumed that the leg lengths are in meters. This experiment was performed in order to determine which leg to torso ratio would be best for the SLIP model robot in both a flat terrain and a rough terrain scenario. All different lengths were tested on both flat and rough terrains and the average, maximum and minimum return of 5 trials over 500 epochs was recorded.

5.3.3 Leg Design

Finally, we performed experiments to explore how the leg design of the robot affected its performance. This allowed us to determine which of the 3 tested

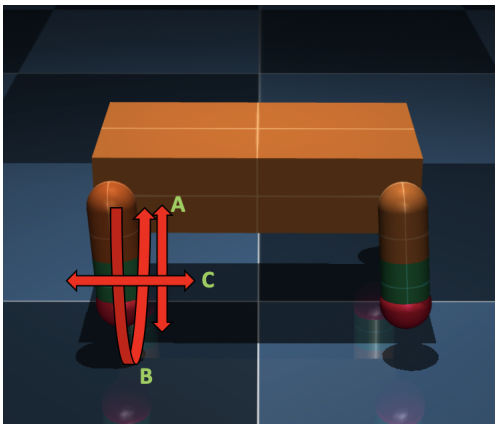


Figure 16: The degrees of freedom on the legs of the SLIP model robot. A: Protraction Retraction (Slide). B: Abduction Adduction (Hip Side). C: Forward and Backward Movement (Hip).

designs should be used in a given situation. We tested the SLIP model legs against 2 different kinds of legs with knee joints in different orientations. The leg designs that were tested can be seen in figure 3. The design shown in the middle is similar to MIT's Cheetah 3 [1] robot, while that on the right is based on ETH's ANYmal [2] and Boston Dynamics Big Dog [4]. All three leg designs were tested on both rough terrain and flat terrain. Again, 5 trials were performed, each over 500 epochs and the average, maximum and minimum return was recorded.

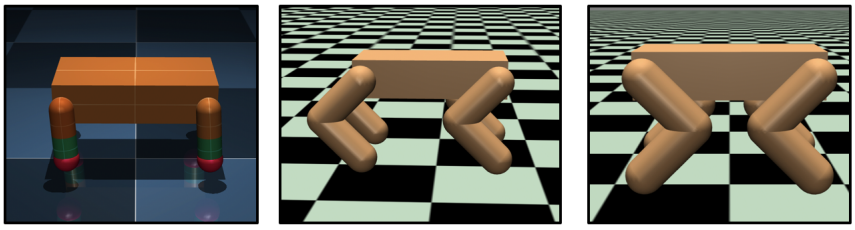


Figure 17: The three types of legs that were tested. Left to right: SLIP model legs, legs with knee joints in the same orientation, legs with knee joints in mirrored orientation

5.4 Experimental Results and Discussion

5.4.1 Degrees of Freedom

The results obtained for the number of degrees of freedom available are shown in figures 4 and 5. As can be seen, the highest median return on flat terrain was achieved with all degrees of freedom (DOF) available, although with the hip degree of freedom (C in figure 2) removed the overall maximum return was achieved. In order to understand how the robot moved forward without the hip degree of freedom, we ran the results visually using MuJoCo and discovered that the robot was using the sideways degree of freedom on the hip to walk laterally instead of forwards and with this was able to achieve a high return. With all degrees of freedom available the robot also does something somewhat unexpected, and uses its front legs for movement and its back legs to stabilize itself with respect to the ground. Since the simulation restarts when the torso collides with the ground, this method was most likely learned in order to avoid contact. It does however manage to move itself forward quite effectively this way, and does not have to be considered as an unsuccessful attempt at learned robot locomotion, as in real life, albeit with some adjustments, this method could be applicable. Further tests would need to be done in order to see if a regular gait as we know it would be able to achieve better returns. In this case some restrictions would have to be put so the algorithm is not able to use the back legs simply for stabilization.

Unlike the flat terrain, on rough terrain the morphology with all degrees of freedom available performed quite poorly. The morphology with the hip degree of freedom removed had the highest median, and that with the slide degree of freedom removed had the overall highest maximum average return. These results are surprising, as it is normally assumed that the higher the number of degrees of freedom in a robot, the more flexible it is to navigate throughout rough terrain. A possible reason for the bad performance of the model with all degrees of freedom available is that the robot is less stable in the initial position on the rough ground since the straight legs that can easily slip away on an uneven surface. This causes the torso of the robot to collide with the ground, in turn concluding the current episode, and therefore impairing learning. Additionally, the straight legs cannot easily pick the body back up when it is close to the ground. The robot with the straight legs therefore has an immediate disadvantage when the simulation starts. To tackle this problem a stability procedure could be implemented in order to start the simulation with the robot standing steadily on the terrain rather than getting dropped on like it is currently.

5.4.2 Leg Lengths

The results obtained when comparing the leg lengths of the SLIP model robot are shown in figures 6 and 7. As shown in figure 6 the highest median return, as well as the overall maximum return on flat terrain was achieved with a leg length

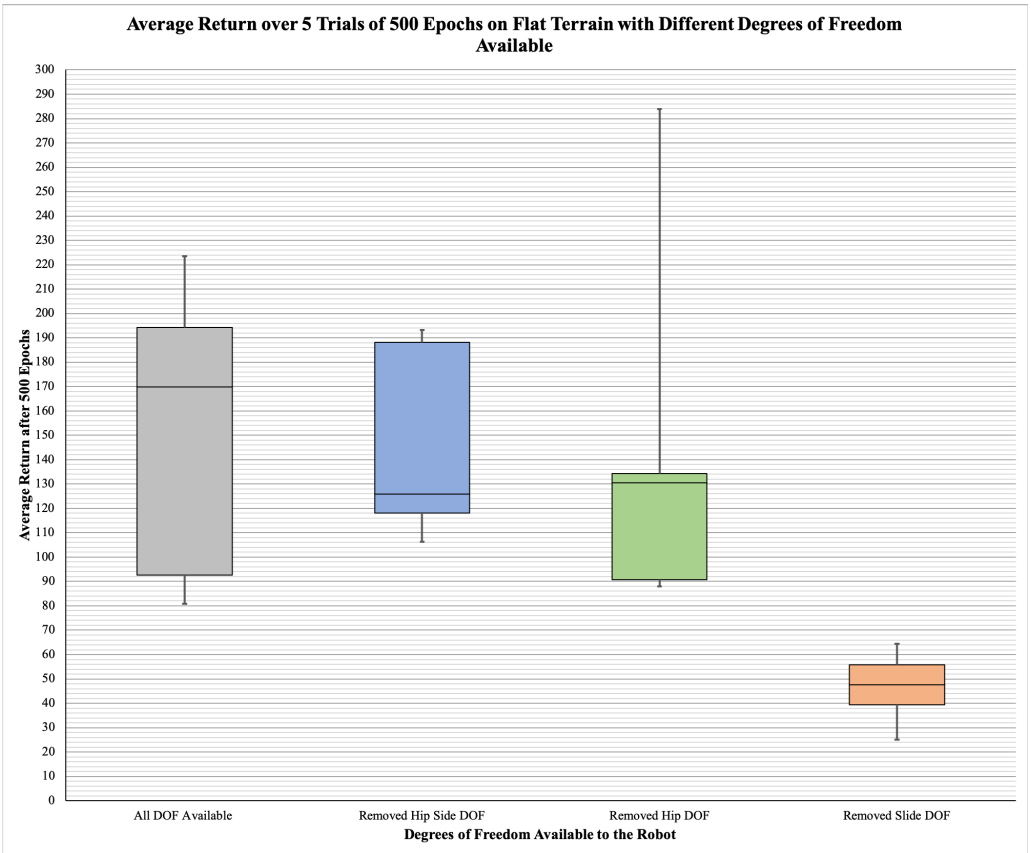


Figure 18: Results for Degrees of Freedom Experiments on the SLIP Model Robot on Flat Terrain

of 0.2. The difference in the average return between the different leg lengths was significant, with a P-value of 0,0778 as calculated using the repeated measures ANOVA test. On rough terrain, the highest median return, as well as the overall maximum return are also achieved at a leg length of 0.2 as shown in figure 7. Again the results differed significantly with a P-value of 0,0090. In both the flat and rough terrain scenarios a leg length of 0.5 performed the worst overall, with the lowest overall median. These results suggest that one ratio of leg length to torso size can be implemented on a quadrupedal robot and perform well on both rough and flat terrain. All robot models have a rectangular torso with ratios 0.4 x 0.2 x 0.1. The ratio of the legs to the length of the torso for a quadrupedal robot with SLIP model legs should therefore be 1:2.

Further experiments were conducted to find if a length of 0.25 performed even

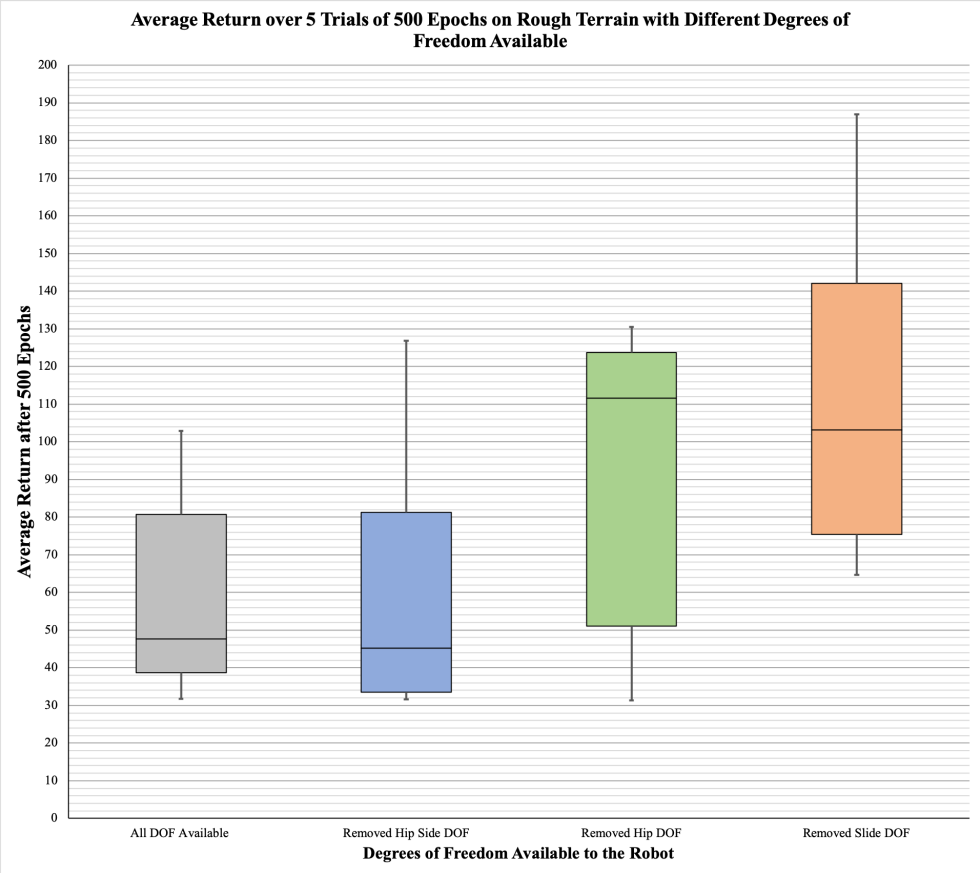


Figure 19: Results for Degrees of Freedom Experiments on the SLIP Model Robot on Rough Terrain

better and it was found that on flat terrain a median of 169.9 could be achieved. For rough terrain the median for a leg length of 0.25 was only 47.6, lower than the median for a length of 0.2. The maximum return achieved however was 102.9, which is higher that the maximum return achieved with a leg length of 0.2 on rough terrain. Additionally, the rough terrain that the experiments were performed on did not have large variations in height as can be seen in figure . It would therefore be recommendable to perform more experiments in order to analyze results over more trials, as well as testing leg lengths over different kinds of rough terrain. For now, it would be desirable for the leg length of the robot to be able to dynamically increase to 0.25 when the robot is walking over flat terrain, while staying at 0.2 in order to traverse a rough terrain scenario.

5.4.3 Leg Design

For the experiments conducted in order to compare different types of leg and their performance, the results can be seen in figure 8 and 9. As shown in figure 8, the straight legs with slide joints (i.e. SLIP model legs) performed best on flat terrain. The results varied significantly between different leg types, as shown by a P-value of 0,0307 calculated using the repeated measure ANOVA test. Conversely, on rough terrain the straight legs performed the worst overall with the lowest median average return, while the knee joints facing in the same direction performed the best. The difference in results here was less significant, with a P-value of 0,10214964. The good performance of the SLIP model legs on flat terrain is to be expected, as on flat terrain it should have no issues with stability at the start, and the three available degrees of freedom give it a large range of motion. After investigating the visualization of the simulation, it seems that the robot does not learn to walk in the expected way. Instead of using all four legs to walk the robot seems to stabilizes itself with its back legs and uses its two front legs to move itself forward. This is to be expected with a learning algorithm, and as discussed in section 6.5.1, steps would need to be taken in further research to prevent the learning from getting stuck in a method like this.

The good performance of the legs with the knee joints facing in the same direction can be attributed partly to the stability of the model at the start of the simulation. With the bent legs and only the hip and knee degree of freedom the robot is able to stably start learning. The straight legs with the extra side degree of freedom can easily slip away when the robot is dropped from a small height onto the terrain at the beginning of the simulation. Additionally the knee degree of freedom allows the robot to bound more easily over larger parts of the terrain, which can be seen occurring in a visualisation of the simulation. This bounding easily causes higher return since return is measured as the distance covered in the x direction per time step. It should also be noted that the performance of the model with the knees facing the same direction is not always very high, with the minimum recorded average return being only 36.94 after 500 epochs. The knee joints facing in opposite directions had overall lower return, but the range between the highest and lowest average return was much smaller, and therefore may be a good alternative to the other kneed legs. Additionally, the rough terrain did not have a large variation in height, and therefore the performances of the legs may vary on other rough terrains. Overall, these results show that a straight SLIP model legs can be recommended for flat terrain scenarios, while knee joints with the knees facing in the same direction can be recommended for rougher terrain scenarios.

5.5 Conclusion and Further Work

Overall some cautious recommendations can be made for which morphology of a quadrupedal robot to employ given a scenario. When the terrain is rough, it is best to have a robot with knee joints facing the same direction as shown the

middle image of figure 3. If a SLIP model robot does get employed, the best length of legs to use is 0.2, given the torso dimensions are 0.4 x 0.2 x 0.1. In other words, the best ratio of leg length to torso length would be 1:2. According to our results, return is also higher when the hip degree of freedom is removed.

On flat terrain, the best model of robot to employ is the straight legged robot with SLIP model legs. The legs should be a length of 0.25 given the torso dimensions are 0.4 x 0.2 x 0.1. In this case the ratio would be 5:8 for leg length to torso length. Finally, having all three degrees of freedom tested available would provide the best results.

These recommendations are based on a lot of assumptions. Firstly, they are all based on the overall performance of the learning algorithm created. With other algorithms, the results may have differed significantly, as bias towards one setup or type may exist within the algorithm. Additionally, since the experiments were performed with the use of a learning algorithm, there could be high variations in performance between each run depending on various factors. This would need to be taken into consideration when running the experiments. Performing many more trials for each experiment may have helped to make the results more reliable, however due to time constraints not many experiments could be conducted. The learning algorithm was run on a laptop with a 3,1 GHz processor and an experiment of 5 trials of 500 epochs took around 3 hours to complete. Additionally, different kinds of rough terrains should be tested in the future, as the experiments for rough terrain were performed using only one type of rough terrain. Good performance on one type of rough terrain does not mean the same on a different type.

In conclusion, many more leg lengths, leg types and terrain types could be tested in the future in order to provide a broader overview of which morphologies are best, however we made a starting contribution to the subject.

6 Acknowledgements

We wish to thank Dr. Rico Möckel for his guidance and support throughout our project. Additionally we would like to express our gratitude towards all members of the SwarmLab, especially Seethu Christopher and Lucas Dahl for their help with the motor controllers and as well as for kindly accommodating us in the lab throughout the duration of the year.

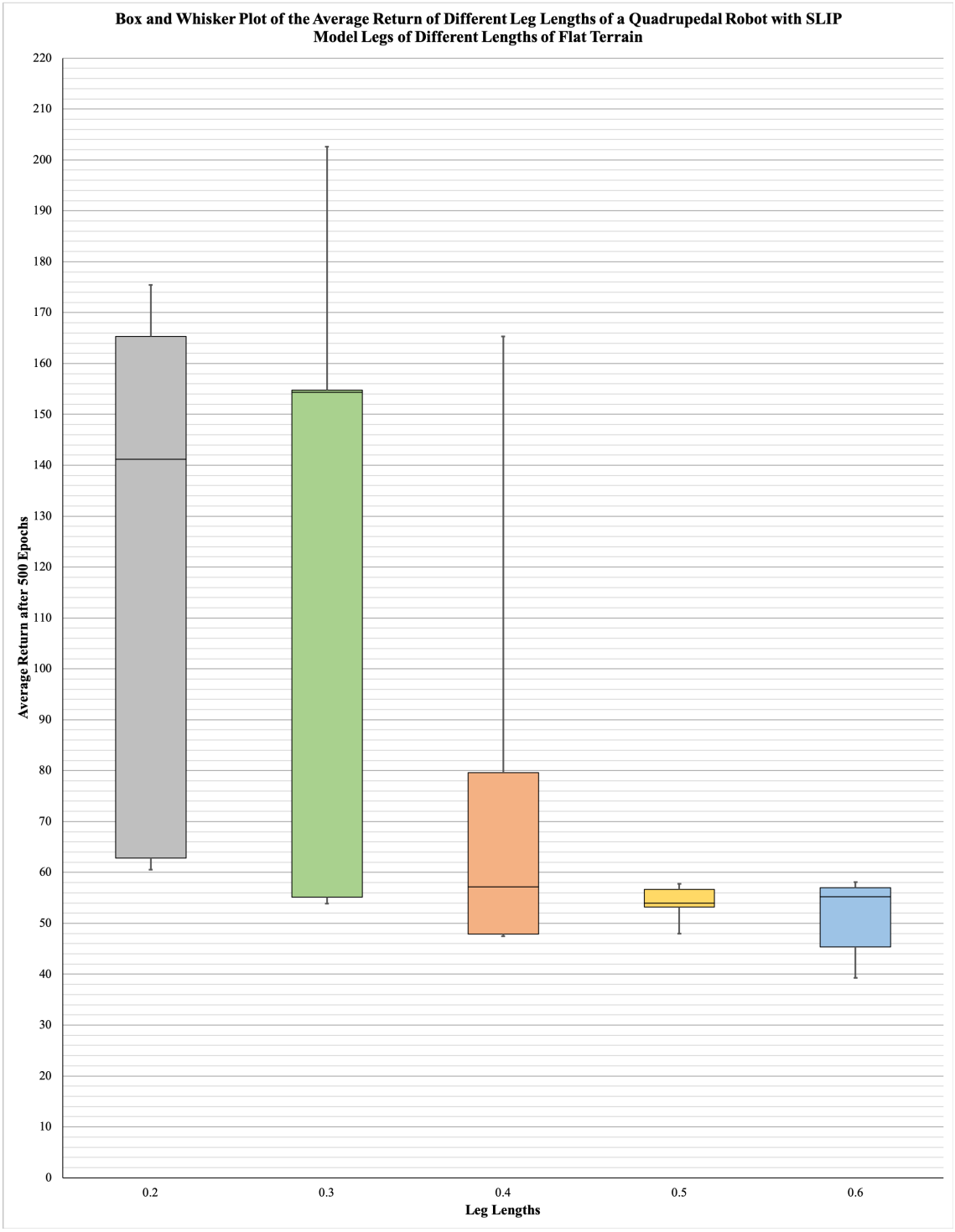


Figure 20: Results for the Leg Length Experiments on Flat Terrain

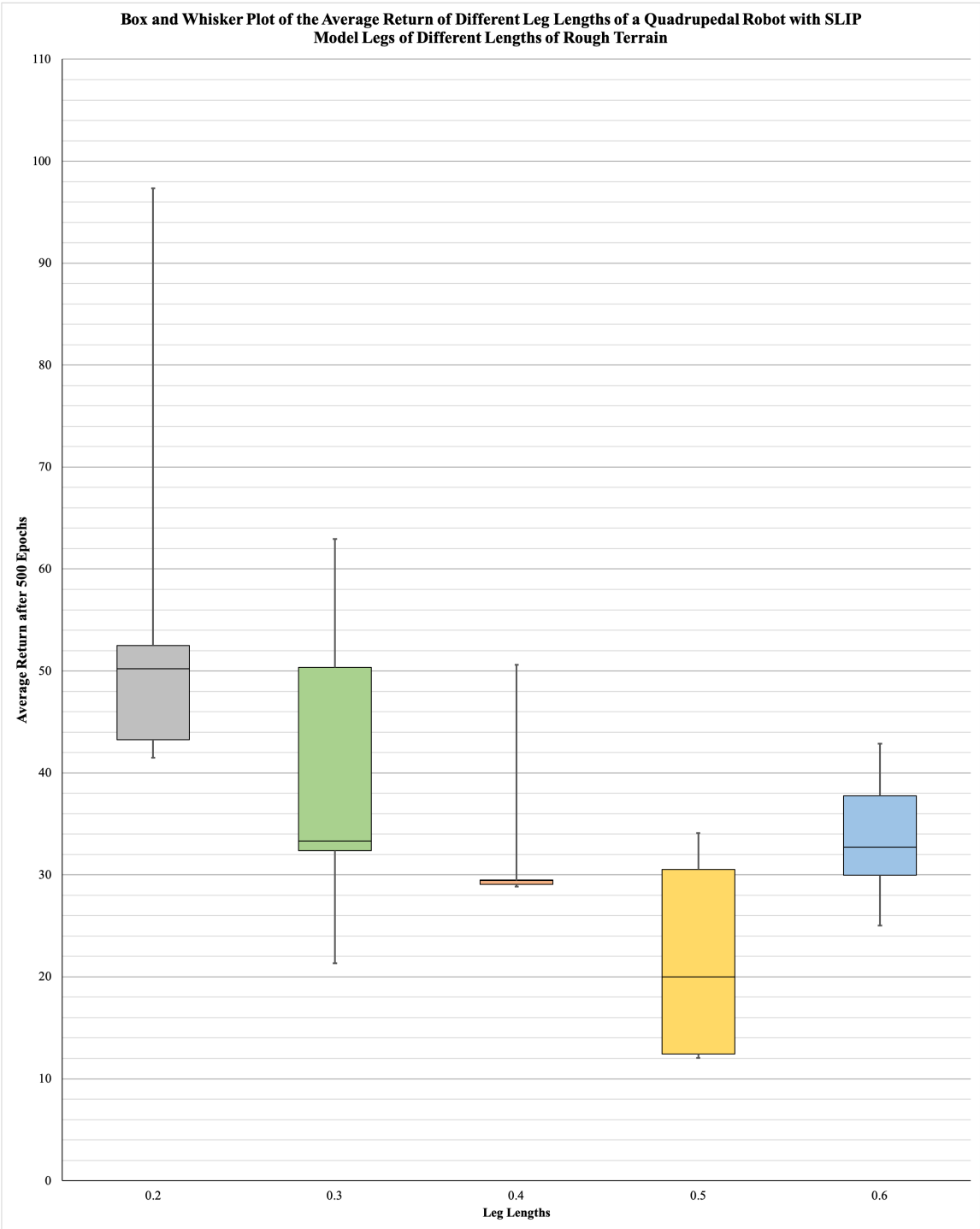


Figure 21: Results for the Leg Length Experiments on Rough Terrain

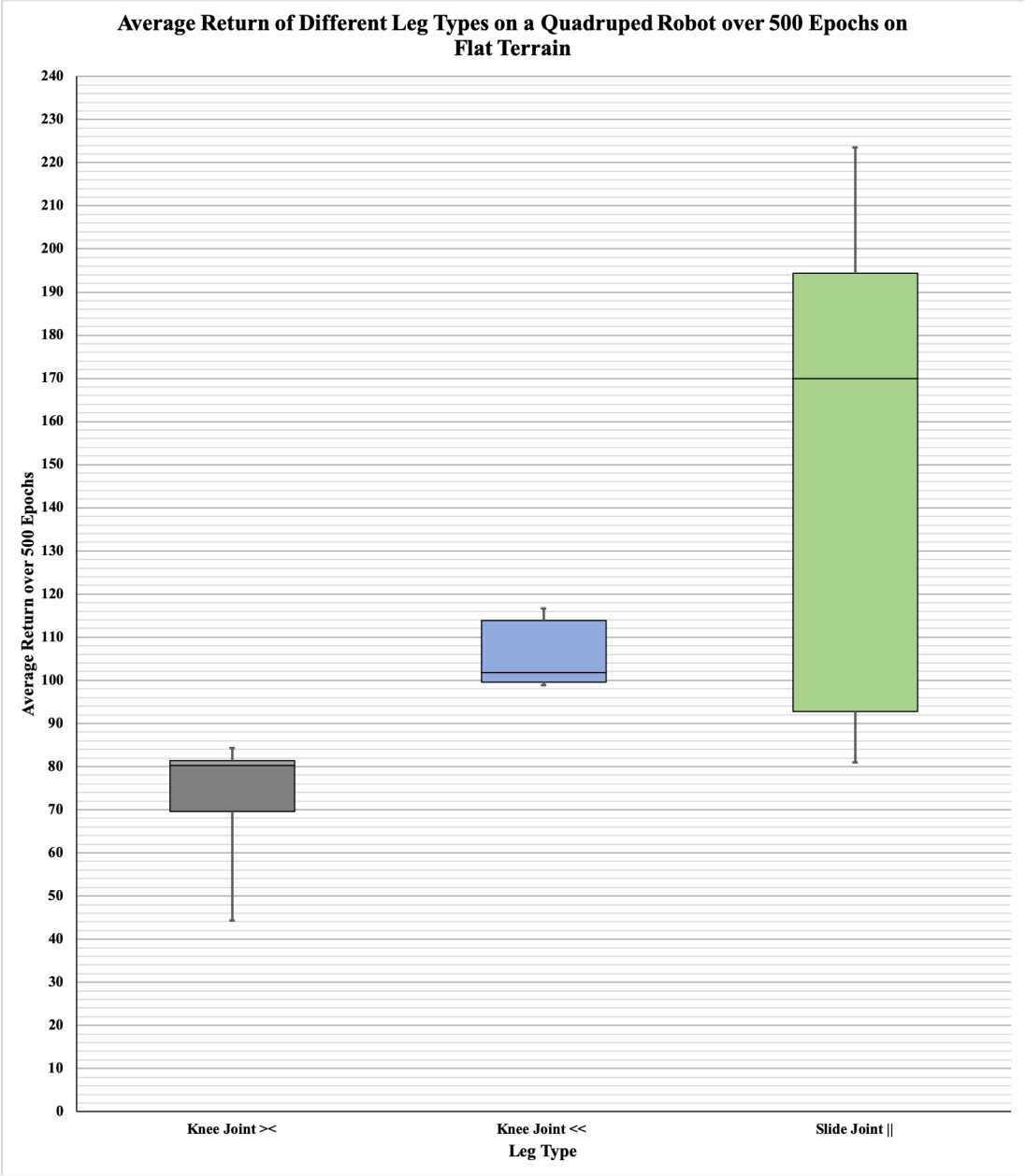


Figure 22: Results for the Leg Type Experiments on Flat Terrain

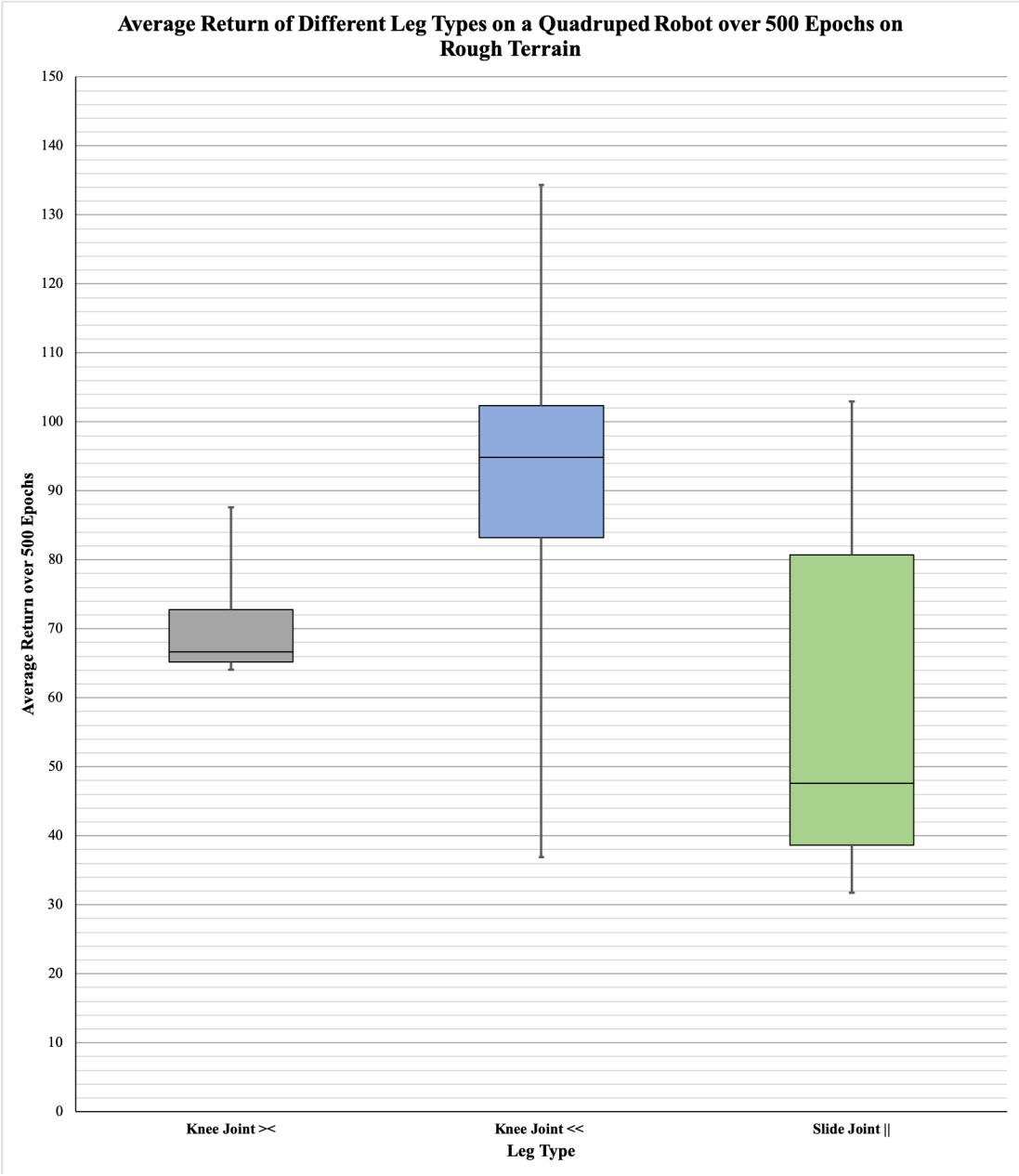


Figure 23: Results for the Leg Type Experiments on Rough Terrain

References

Aoyama, T., Sekiyama, K., Hasegawa, Y., & Fukuda, T. (2009). Optimal limb length ratio of quadruped robot minimising joint torque on slopes. *Applied Bionics and Biomechanics*, *6*(3-4), 259–268.

Barasuol, V., Buchli, J., Semini, C., Frigerio, M., De Pieri, E. R., & Caldwell, D. G. (2013, May). A reactive controller framework for quadrupedal locomotion on challenging terrain. In *2013 ieee international conference on robotics and automation* (p. 2554-2561). doi: 10.1109/ICRA.2013.6630926

Barasuol, V., Buchli, J., Semini, C., Frigerio, M., De Pieri, E. R., & Caldwell, D. G. (2013). A reactive controller framework for quadrupedal locomotion on challenging terrain. In *2013 ieee international conference on robotics and automation* (pp. 2554–2561).

Beer, R. D., & Gallagher, J. C. (1992). Evolving dynamical neural networks for adaptive behavior. *Adaptive behavior*, *1*(1), 91–122.

Bellicoso, C. D., Bjelonic, M., Wellhausen, L., Holtmann, K., Günther, F., Tranzatto, M., ... Hutter, M. (2018). Advances in real-world applications for legged robots. *Journal of Field Robotics*, *35*(8), 1311–1326.

Bledt, G., Powell, M. J., Katz, B., Di Carlo, J., Wensing, P. M., & Kim, S. (2018). Mit cheetah 3: Design and control of a robust, dynamic quadruped robot. In *2018 ieee/rsj international conference on intelligent robots and systems (iros)* (pp. 2245–2252).

De Santos, P. G., Cobano, J. A., Garcia, E., Estremera, J., & Armada, M. (2007). A six-legged robot-based system for humanitarian demining missions. *Mechatronics*, *17*(8), 417–430.

Duan, Y., Chen, X., Houthooft, R., Schulman, J., & Abbeel, P. (2016). Benchmarking deep reinforcement learning for continuous control. In *International conference on machine learning* (pp. 1329–1338).

Geyer, H. (2005). *Simple models of legged locomotion based on compliant limb behavior= grundmodelle pedaler lokomotion basierend auf nachgiebigem beinverhalten* (Unpublished doctoral dissertation).

Group, E. T. (2018). *Ethercat brochure.* `https://www.ethercat.org/`. Author.

Haarnoja, T., Zhou, A., Ha, S., Tan, J., Tucker, G., & Levine, S. (2018). Learning to walk via deep reinforcement learning. *arXiv preprint arXiv:1812.11103*.

Heess, N., Sriram, S., Lemmon, J., Merel, J., Wayne, G., Tassa, Y., ... others (2017). Emergence of locomotion behaviours in rich environments. *arXiv preprint arXiv:1707.02286*.

Hutter, M., Gehring, C., Jud, D., Lauber, A., Bellicoso, C. D., Tsounis, V., ... others (2016). Anymal-a highly mobile and dynamic quadrupedal robot. In *2016 ieee/rsj international conference on intelligent robots and systems (iros)* (pp. 38–44).

IgH. (2013). *Igh ethercat master documentation.* `https://etherlab.org/en/ethercat/`.

Ijspeert, A. J. (2008). Central pattern generators for locomotion control in animals and robots: a review. *Neural networks*, *21*(4), 642–653.

Khoramshahi, M., Spröwitz, A., Tuleu, A., Ahmadabadi, M. N., & Ijspeert, A. J. (2013). Benefits of an active spine supported bounding locomotion with a small compliant quadruped robot. In *2013 ieee international conference on robotics and automation* (pp. 3329–3334).

Kolter, J. Z., Rodgers, M. P., & Ng, A. Y. (2008). A control architecture for quadruped locomotion over rough terrain. In *2008 ieee international conference on robotics and automation* (pp. 811–818).

Lee, J., Hwangbo, J., & Hutter, M. (2019). Robust recovery controller for a quadrupedal robot using deep reinforcement learning. *arXiv preprint arXiv:1901.07517*.

Levine, S., Pastor, P., Krizhevsky, A., Ibarz, J., & Quillen, D. (2018). Learning hand-eye coordination for robotic grasping with deep learning and large-scale data collection. *The International Journal of Robotics Research*, *37*(4-5), 421–436.

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., ... Wierstra, D. (2015). Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*.

Mahmood, A. R., Korenkevych, D., Vasan, G., Ma, W., & Bergstra, J. (2018). Benchmarking reinforcement learning algorithms on real-world robots. *arXiv preprint arXiv:1809.07731*.

Matsubara, T., Morimoto, J., Nakanishi, J., Sato, M.-a., & Doya, K. (2006). Learning cpg-based biped locomotion with a policy gradient method. *Robotics and Autonomous Systems*, *54*(11), 911–920.

Michel, O. (2004). Cyberbotics ltd. webots$^{\mathrm{TM}}$: professional mobile robot simulation. *International Journal of Advanced Robotic Systems*, *1*(1), 5.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

Mori, T., Nakamura, Y., Sato, M.-A., & Ishii, S. (2004). Reinforcement learning for cpg-driven biped robot. In *Aaai* (Vol. 4, pp. 623–630).

Motor, M. (2015). *Maxpos 50/5 firmware specification.* `https://maxonmotor.com/`.

Nagabandi, A., Kahn, G., Fearing, R. S., & Levine, S. (2018). Neural network dynamics for model-based deep reinforcement learning with model-free fine-tuning. In *2018 ieee international conference on robotics and automation (icra)* (pp. 7559–7566).

Nakanishi, J., Morimoto, J., Endo, G., Cheng, G., Schaal, S., & Kawato, M. (2004). Learning from demonstration and adaptation of biped locomotion. *Robotics and autonomous systems*, *47*(2-3), 79–91.

Nygaard, T. F., Martin, C. P., Torresen, J., & Glette, K. (2018). Self-modifying morphology experiments with dyret: Dynamic robot for embodied testing. *arXiv preprint arXiv:1803.05629*.

OpenAI. (2018). *spinningup.* `https://github.com/openai/spinningup`. GitHub.

Peng, X. B., Berseth, G., & Van de Panne, M. (2016). Terrain-adaptive locomotion skills using deep reinforcement learning. *ACM Transactions on Graphics (TOG)*, *35*(4), 81.

Peng, X. B., Berseth, G., Yin, K., & Van De Panne, M. (2017). Deeploco: Dynamic locomotion skills using hierarchical deep reinforcement learning. *ACM Transactions on Graphics (TOG)*, *36*(4), 41.

Peng, X. B., & van de Panne, M. (2017). Learning locomotion skills using deeprl: Does the choice of action space matter? In *Proceedings of the acm siggraph/eurographics symposium on computer animation* (p. 12).

Schulman, J., Levine, S., Abbeel, P., Jordan, M., & Moritz, P. (2015). Trust region policy optimization. In *International conference on machine learning* (pp. 1889–1897).

Schulman, J., Moritz, P., Levine, S., Jordan, M., & Abbeel, P. (2015). High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*.

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., ... others (2017). Mastering the game of go without human knowledge. *Nature*, *550*(7676), 354.

Tan, J., Zhang, T., Coumans, E., Iscen, A., Bai, Y., Hafner, D., ... Vanhoucke, V. (2018). Sim-to-real: Learning agile locomotion for quadruped robots. *arXiv preprint arXiv:1804.10332*.

Todorov, E., Erez, T., & Tassa, Y. (2012). Mujoco: A physics engine for model-based control. In *2012 ieee/rsj international conference on intelligent robots and systems* (pp. 5026–5033).

Whitehead, S. D. (1991). A complexity analysis of cooperative mechanisms in reinforcement learning. In *Aaai* (pp. 607–613).

7 Appendix

7.1 Physics Simulation [Kaspar]

At first we used the MuJoCo (Multi-Joint dynamics with Contact) simulator, developed by Roboti LLC, [39] to simulate the quadruped robot. However, the MuJoCo academic license is not free, so instead we opted for Webots, which has a free academic license. Webots is a simulation environment made by Cyberbotcis Ltd for simulating mobile robots.

MuJoCo has its own custom physics engine, whereas Webots uses ODE (www.ode.org). The MuJoCo engine in general is fast and accurate, but it slows down as more disconnected bodies are introduced (). ODE is an open-source physics engine. It is widely used in robotics applications such as Gazebo. ODE can be slower than the MuJoCo engine when handling simpler scenarios, but in some cases where there are for example many disconnected bodies ODE is able to perform better and faster (sim. tools paper reference, Erez). For our purposes, both engines are suitable.

Both simulators enable the user to make their own models. In MuJoCo modeling is done by by writing an XML file in MuJoCo's MJCF format (Todorov, 2012). In webots, the VRML97 (Virtual Reality Modelling Language) standard is used to define 3D models (webots reference, Michel, 2004). VRML allows you to define basic hierarchies of primitive solids such as cylinders and cubes. Webots requires world files (.wbt) to run a simulation, in world files VRML models are used to represent desired models.

Webots is set out as an OOP interface for robot control. Various devices in Webots such as gyroscopes and distance sensors correspond to objects. MuJoCo is not organized in an OOP way. Instead, the simulation process is divided into a small number of data structures on which operations are performed: the model and the data. The model has the needed details to describe the physical system and the data contains reusable intermediate solutions of computations and other information on the state.

By default, Webots has slightly more sensors than MuJoCo, but both simulators have sufficiently many sensors for our purposes. Some useful sensors include: GPS, gyroscope and inertial unit. Webots also has a slightly better selection of actuators than MuJoCo, but again, for our purposes both simulator's actuators are sufficient.

MuJoCo has a minimalist user interface- it works, but there are no extra details. For our purposes it would be helpful to have a more detailed user interface. Webots provides a good user interface. It has an interface with which you can easily control the simulation, e.g. fast forward, advance simulation by one step, etc. Webots' user interface also provides you with a lot more functionality and options than MuJoCo' user interface. For example, Webots provides the user with a text editor within the simulator with which you can edit robot controllers. It is also very helpful that Webots allows you to access and edit the currently running simulation's world file during simulation, so it is easy to visually see how your changes affect the simulation. Webots does

not let the user run any simulations without the Webots simulation window being open. This can be problematic, for example in our case many simulations are to be run at the same time. Ideally Webots would enable the user to run simulations without needing to have the simulation window open, but in its current state, running many simulations simultaneously also causes multiple Webots simulation windows to pop up. MuJoCo on the other hand offers more flexibility in this regard, enabling the user to run simulations without the need to have the simulation window open.

Both simulators have desirable properties and both would be sufficient for our purposes. Webots was chosen in the end as it has a free academic license. It is also more intuitive to use as a beginner when it comes to using simulators. It is easier to understand Webots thanks to its helpful user interface. The developers of the Webots simulator also have an online support chat which is very helpful.

7.2 Webots Model Tables

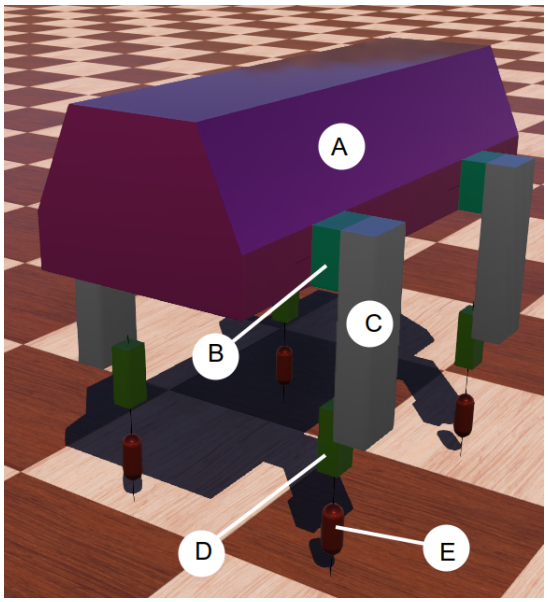


Figure 24: The quadrupedal model designed in Webots. Its parts are labelled by letters in the order in which the parts appear in the node hierarchy. A: Torso, B-D: Leg, E: Foot. The foot (part E) is connected to the leg by a spring, which is represented by slim black lines.

Table 1: Properties of the Webots quadrupedal model's solids

Tables 2-4: Properties of the Webots model's motors. These tables are connected, one table continues from the previous table.

Component	Shape Used	Mass (kg)
A	Indexed face set	12.0
B	Box	0.03
C	Box	0.17
D	Box	0.15
E	Cylinder	0.1

Motor	Type	Acceleration(rad/sˆ(2))	Maximum Velocity (m/s)
Between parts A and B	Rotational	1.5	3
Between parts C and D	Linear	0.75	0.2

7.3 Motor Control [Mathieu]

The motors that were used are the EC-4pole 22 brushless DC motors made by Maxon Motors. The motor controllers that were used are the Maxpos 50/5 motion controllers made by Maxon Motors. Communication with the motor controllers happens through the EtherCAT (Ethernet for Control Automation Technology) fieldbus, this allows for short cycle times of less than 100μs and low jitter of less than 1μs, making them very suitable for robotics applications[10]. The Maxpos 50/5 is an EtherCAT Slave device. The slave devices can be connected in sequence. Slave devices are controlled by a master. The master device sends out an EtherCAT Telegram that passes through all the slave devices. When the telegram passes through a slave device it reads the data addressed to it and inserts its data into the telegram as it is moves along, once the last node is reached, the data is send back to the master. The master device is the only node that which actively sends a data frame, all the other nodes only pass it along.

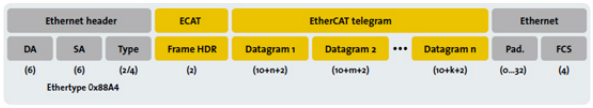


Figure 25: EtherCAT Frame

The master is implemented as a software solution. The master that was used is the IgH EtherCAT Master for Linux, which runs as a Linux kernel module[14]. The operating system that was used is Ubuntu 16.04.6 with a kernel that was patched with the PREEMPT_RT patch. Communication with the master happens via the IgH EtherCAT Master's C API. It allows for the separation of slave groups through domains and is therefore well suited for the task of robotic control making it easy to independently control each joint or simultaneously control groups of joints. The motor controller supports several operating modes. The operating mode that was used is Cyclic Synchronous Position Mode (CSP). With CSP, the master provides a target position to the drive in a cyclic synchronous manner[26].

Motor	Maximum Force (N)	Torque (N*m)
Between parts A and B	N/A	10
Between parts C and D	58.5	N/A

Motor	Min. Pos. (linear-m, rotational-rads)	Max. Pos. (lin.-m, rot.-rads)
Between parts A and B	-3.14	3.14
Between parts C and D	-0.01	0.085

7.4 Learning on Different Terrains - Extra Graphs Regarding the Experiments

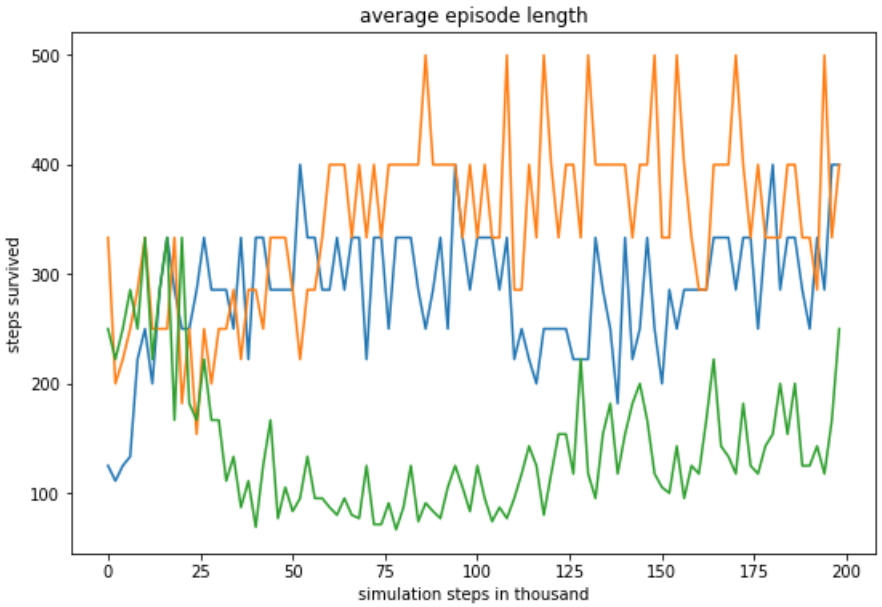


Figure 26: Easy elevation grid episode length

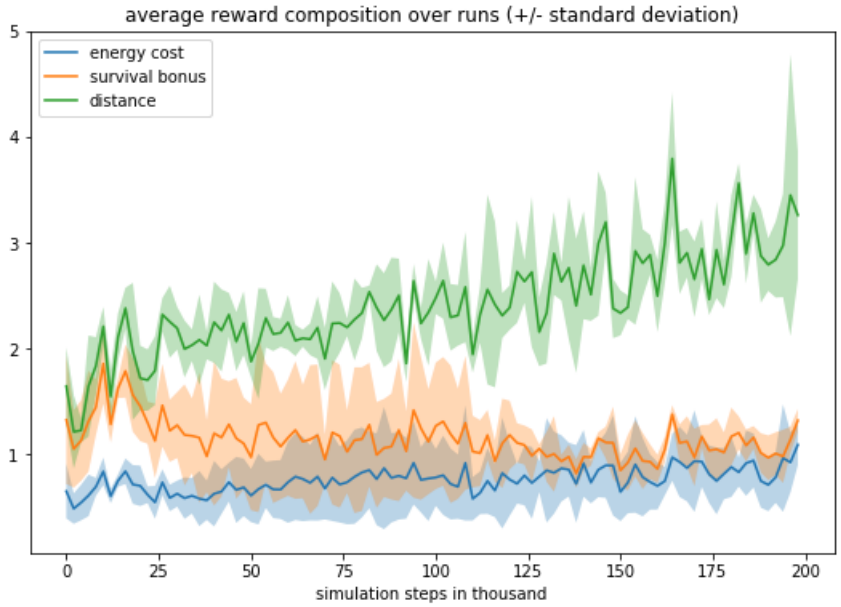


Figure 27: Easy elevation grid Average Reward Composition

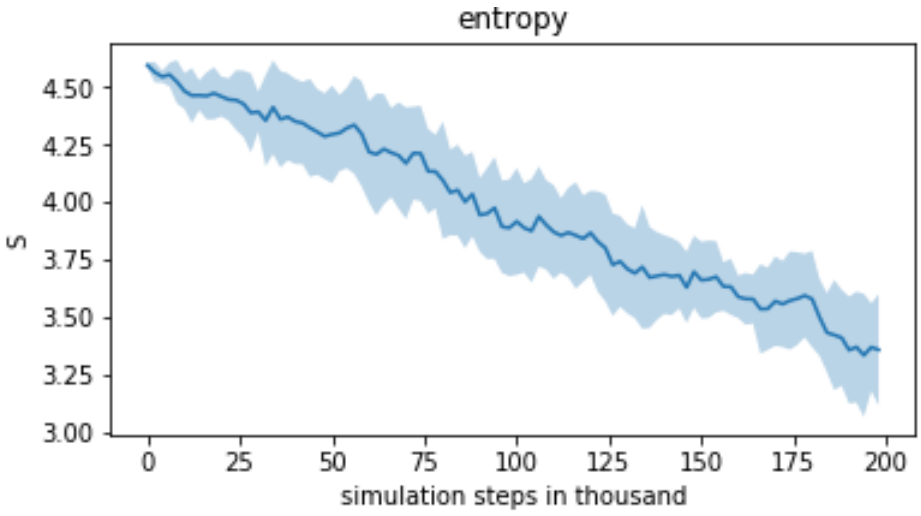


Figure 28: Easy elevation grid entropy

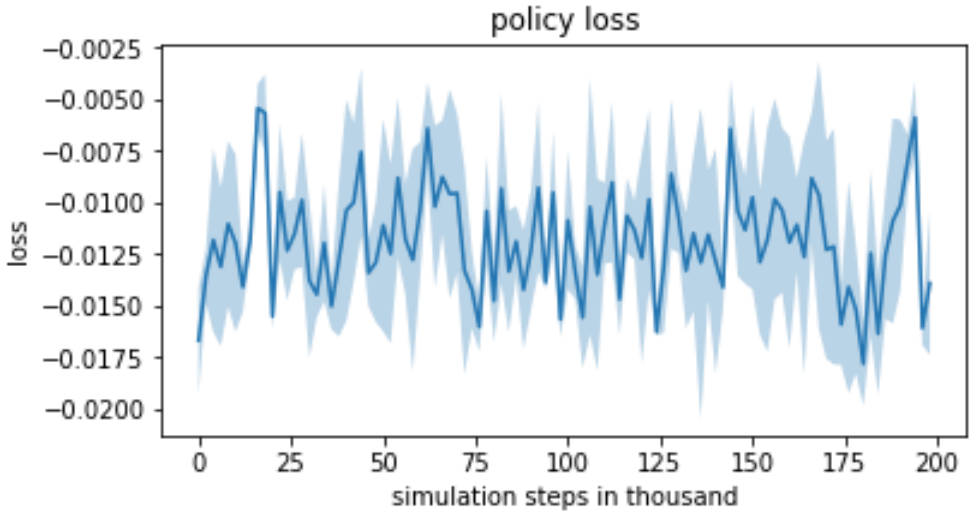


Figure 29: Easy elevation grid policy loss

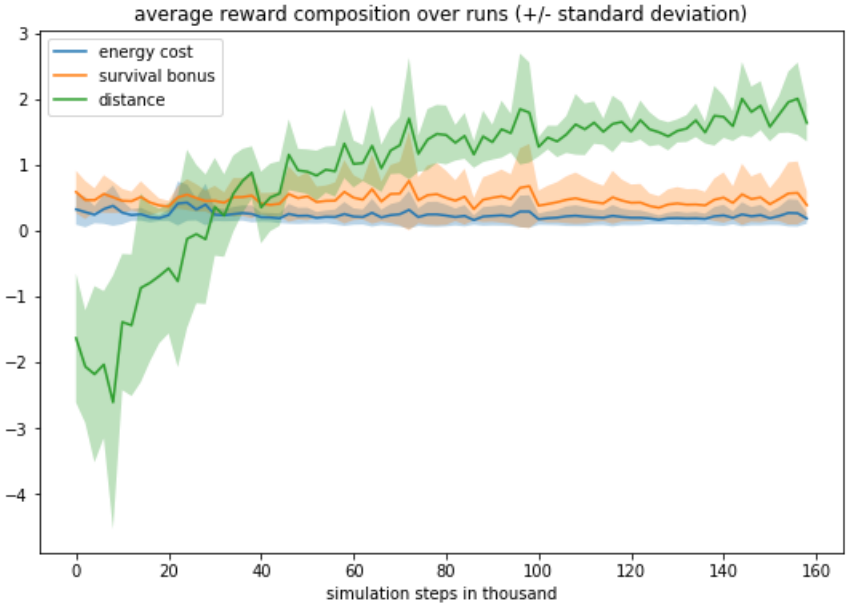


Figure 30: Hard elevation grid Average Reward Composition

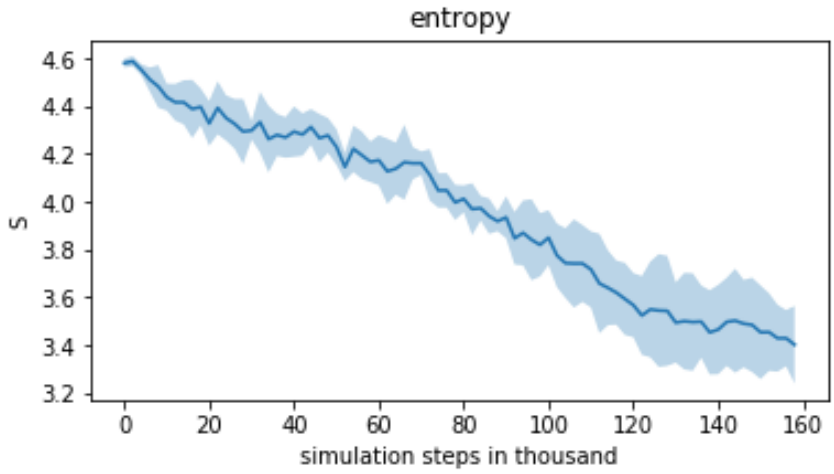


Figure 31: Hard elevation grid entropy

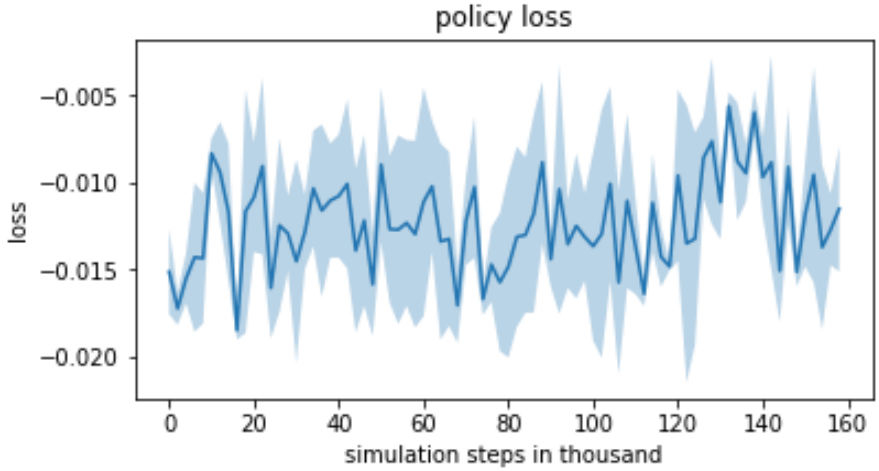


Figure 32: Hard elevation grid policy loss