SpotMicroAI: a micro step towards intelligent locomotion

Leon Eshuijs (11866071)
University of Amsterdam
Science Park 900

1 Introduction

Legged robotics offer important advantages over their wheeled counterparts for real-life applications. Due to their ability to have a small selection of contact points, they are better capable to pass obstacles, making them much more suitable to thread through rough terrain. Until recent years, research on learning-based methods for locomotion has mostly been limited to simulation. One of the primary reasons research on real-world systems has remained marginal is that learning on real robots is both slower and more expensive than simulation training. Where simulations can be reset if models fail, on the real robots failure can result in the destruction of expensive robotics components. The cost component is exacerbated by the nature of learning-based approaches, which require a lot of exploratory movements during the early stages of learning, which increases the chances of damaged components.

When learning-based methods are applied to real robots, a common practice is the use of robots that contain state-of-the-art actuators and sensors [Hwangbo et al., 2019; Lee et al., 2020], such as the ANYmal [Hutter et al., 2016]. Although high-end components increase the accuracy of the actuator control, it limits research to well-funded research groups. Recent work by [Rahme et al., 2020b], however, showed that learning-based models can be robust enough to apply to hobby robotics. By randomizing the dynamics of the robot and the environment in simulation, their approach named D^2-GMBC bridged the sim-to-real gap on a variation of the open-source robotics project *SpotMicroAI*[1]. However, the used Reinforcement Learning (RL) algorithm called Augmented Random Search (ARS) consists of only a single-layer Neural Network.

In this report, the effect of more complex RL techniques on the D^2-GMBC framework is evaluated. To this end, the Soft Actor-Critic (SAC) algorithm from [Haarnoja et al., 2018a] is applied, since its underlying entropy maximizing approach has been shown to provide both robust models and fast learning. We evaluate the two approaches in three ways. Firstly, by comparing the learning trajectory of the agents during simulation training. Secondly, by comparing the walking speed of the agents on the real robot and

[1] https://spotmicroai.readthedocs.io/en/latest/

comparing their performance against the fixed gait situation. Lastly, a broader evaluation is performed on video material of the gaits to understand the benefits and the downsides of the agents.

The experiments show a much lower reward for the ARS agent than for the SAC agent. However, since the ARS agent is able to leverage parallel roll-outs it is not clear if this performance is mainly due to the increased computation. From the experiments on the real robot we found that although neither of the agents was faster than the baseline gait, the RL agents did display dynamical falling prevention behavior. This behavior was most visible in the SAC agent, but also seemed to be the cause of the reduced traversal speed.

1.1 Related work Background

In this section, a short overview will be given of the research area that is involved with transferring policies learned in simulation to the real world, known as sim-to-real. For each work, we will lay out important learning aspects for locomotion such as the observation space, action space, RL agent, as well as other training parameters that are important for the results of the papers.

A popular approach to limit the sim-to-real gap focuses on increasingly accurate simulations. One of the most notable successes came from [Hwangbo et al., 2019], which applied simulation pretraining for real-world quadrupled locomotion. Applied on the ANYmal robot, a dog-sized quadrupled, an RL policy was learned to output the desired joint state of each actuator. To achieve accurate simulation training, the actuators in the simulation are modeled by a separate neural network that is trained beforehand on joint encoder data from the real actuators. Their trained model allowed the ANYmal robot to walk more efficiently, increasing the walking speed by 25% compared to previous implementations.

Because high-precision modeling of the actuators is expensive and often unattainable, the work of [Peng et al., 2018] proposes an alternative method. Instead of handpicking the right actuator characteristics in simulation, or training a separate network to model the actuator, their work demonstrated that randomizing the dynamical characteristics of robotics systems can increase robustness. Hereby, the policy is trained to work under a wide range of dynamics, which reduces overfitting on the simulation, and allows for better adaptation to the dynamics of the real robot.

Based on this work, [Tan et al., 2018] apply dynamics randomization and perturbation forces during the training of the Minitaur robot in simulation. The policy network is trained using Proximal Policy Optimization (PPO), an RL algorithm that has been shown to increase on-policy stability and allow for easy parallelization [Hafner et al., 2017]. The model returns the eight motor angles, and the reward is chosen to increase the speed whilst reducing high energy consumption, to create the optimal gait. Lastly, the authors identify two other components to reduce the sim-to-real gap. Firstly, they find that the latency between sending the motor command and receiving the sensory change of this action is of great importance and thus worth fine-tuning. Secondly, the authors find that their model increased in robustness by reducing the observation space. This was achieved by removing the sensory data from the joint encoder, and only taking as observation the data from the Internal Measurement Unit (IMU). The IMU sensor provides information about the orientation and accelerations of the body.

Instead of pretraining in simulation, the work of [Haarnoja et al., 2018a] addresses the problems of real-world training by using a sample-efficient deep RL network based on the Soft Actor-Critic (SAC) algorithm of [Haarnoja et al., 2018b]. By training only on the real robot, their model achieves a stable gait from scratch within 2 hours. SAC is an RL algorithm that minimizes the maximum entropy and has been shown to produce robust policies. However, SAC contains a temperature parameter that determines the level of exploration, and in the original implementation of [Haarnoja et al., 2018b] it needs to be manually tuned. To circumvent this tuning, the temperature parameters are taken as a hyperparameter and optimized using dual gradient descent. Although this optimization process does not guarantee to converge to a global optimum, empirical evidence substantiates the efficacy of their method. Similar to

[Tan et al., 2018], the Minitaur is used and the model takes as input the IMU values and joint encoder sensory information. The action space, however, consists of the swing angle and extension of each leg, which are used to compute the joint states of the actuators through inverse kinematics.

More recently [Lee et al., 2020] introduced a novel approach to increase the robustness of locomotion on the ANYmal robot. Where previous models only take the current state as observation, their work used a Temporal Convolutional Network to leverage the history of proprioceptive observations. To address the difficulty of learning locomotion policies in rough terrains, privileged learning was applied to reduce the sparsity of the supervised signal. Hereby the policy of the student was trained to align with the policy of the teacher, which has privileged information about the environment not available in the real world. During inference on the real robot, only the student policy is used to control the motion. Their action space follows the work of Policies Modulating Trajectory Generators (PMTG) as introduced in [Iscen et al., 2018]. In the PMTG algorithm, a baseline gait is provided and the agent outputs a set of actions that modify this gait in different ways.

Inspired by the popularity of the open-source project SpotmicroAI[2], the work of [Rahme et al., 2020b] applies a sim-to-real framework on a home-build, slightly modified version of the Spotmicro. Their method uses Dynamics and Domain Randomization for Gait Modulation with Bezier Curves (D^2-GMBC). By using these two types of randomization and reducing the observation to only the IMU data, a more than 4x higher survival rate is achieved on the real robot, compared to fixed open-loop gait. Contrary to PMTG which modifies sine trajectories, D^2-GMBC, extends gait modulation to a broader class of gait trajectories, by modifying Bezier Curves. The policy consists of a simple single-layer neural network that is trained using Augmented Random Search (ARS). By using a simple agent they highlighted the significance of their D^2 approach. However, an important question that remains is how the D^2 competes with more complex RL algorithms.

This work builds upon D^2-GMBC and researches the importance of RL algorithm complexity to the sim-to-real gap of gait modulation methods for locomotion learning. From previous work, [Sutton and Barto, 2018], it is apparent that more complex RL algorithms can increase robustness. At the same time, other works found that reducing the complexity of the model by reducing the observation space also decreases the sim-to-real gap. In this report, we inspect the influence of the complexity of RL algorithms on the sim-to-real gap. To achieve this the SAC algorithm from [Haarnoja et al., 2018a] is applied to model the framework of D^2-GMBC, and compared against ARS.

2 Method

2.1 Problem Statement

The locomotion task of the quadrupled is formulated as a Partially Observable Markov Decision Problem (POMDP). A POMDP is defined by a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{O}, r(s,a), P(s\prime|a,s))$, consisting of the state space $\mathcal{S}$, the action space $\mathcal{A}$, the observation space $\mathcal{O}$, a reward function $r(s,a)$ and the transition probability $P(s\prime|a,s)$. The policy of the model, π as parameterized by the learnable parameters θ, is then tasked to find the best action for each state to maximize the expected reward. For each observation $o \in \mathcal{O}$, the agent takes the action $a \in \mathcal{A}$ based on the policy, formalized as $a = \pi_\theta(o)$. Because the model is applied to the gait modulation task, the current state of the gait is included in this observation space.

2.2 Gait Modulation

The open-loop gait from D^2-GMBC is used and computes a trajectory path for each foot using 12-point Bezier curves. Their model, based on the work of [Hyun et al., 2014], combines multiple 2D gaits into one 3D gait, which allows for flexible control of cyclical movements of the legs' extremities. As depicted in Figure 2, the 2D curve of each foot contains two phases, the swing (purple), when the leg is lifted into the air, and the stance (red), when the foot remains in contact with the ground. For each foot

[2] https://spotmicroai.readthedocs.io

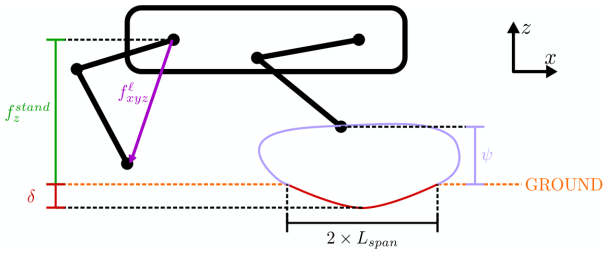


Figure 2: The 2D gait and foot placement parameters from Rahme et al. [2020b]

$l \in \{FL, FR, BL, BR\}$[3], these phases are determined by the phase generator $S^l(t) \in [0, 2)$, where the leg is in stance for $S^l(t) \in [0, 1)$, and swing for $S^l(t) \in [1, 2)$.

To adjust this open-loop gait a set of parameters is provided. The first two parameters are scalars that influence the trajectory of all legs. These are the clearance height ψ, which determines how high the leg lifts during the swing phase, and the virtual ground penetration δ, which is responsible for how high the respective shoulder of each leg is lifted during the stance phase. To adjust the position of the individual legs, a 3-dimensional residual position is defined by $\Delta f_{xyz}^{l} \in \mathbb{R}^3$, resulting in the total residual set Δf_{xyz}. The residual position adjusts the 3D foot position away from the current place in space as determined by the Bezier curve. The policy returns these parameters to adjust the underlying Bezier gait:

$$\Delta f_{xyz}, \psi, \delta = \pi_\theta \left(o_t \right) \tag{1}$$

The last input for the Bezier curve generator Γ, are the external motion commands $\zeta = [\rho, \bar{\omega}, L_{span}]$. From these inputs, L_{span} defines the stride length, ρ rotational angle relative to its momentum, and $\bar{\omega}$, the yaw velocity. During training the motion commands are fixed for $L_{span} = 0.034$ and $\rho = 0$, and an $\bar{\omega}$ is computed in real-time so that the direction of the robot is always straight. The curve generator determines the foot positions by:

$$f_{xyz}^{tg} = \Gamma(S(t), \zeta, \psi, \delta) \tag{2}$$

After adding the residuals, the final position of all the foot endpoints is:

$$f_{xyz} = f_{xyz}^{tg} + \Delta f_{xyz} \tag{3}$$

After the foot positions have been determined, Inverse Kinematics (IK) is used to calculate the joint angles for each of the three actuators of a leg. Firstly note that the foot position from the gait generator Γ, is the position with respect to the shoulder corresponding to each leg. The IK then determines what angle each actuator of a leg should have to achieve this position, under the range constraints of the actuators.

2.3 Observation

Since we can only observe a small set of parameters that determines the current state of the robot, the problem is considered a Partial Observable Markov Decision Problem (POMDP). For the observation space, only the sensory data from the IMU of the robot is used. By keeping the observation to a minimum, we hope to reduce overfitting and ease hardware/sensory restrictions on the real robot. The data taken from the IMU consists of the roll r and the pitch p angles, the 3D angular velocities ω, and the 3D linear accelerations v. The yaw measurements are excluded because they tend to drift fast [Haarnoja et al., 2018a]. The last part of the observation consists of the current internal phase of each leg and is thus not dependent on external factors of the environment. In total, the observation space consists of the 12-dimensional tuple $[r, p, \omega, v, S(t)]$. As mentioned above the action space consists of 14 dimensions, namely the 3D foot position residual of each leg, and the shared scalars delta and psi.

[3]Corresponding to the legs: front left, front right, back left, back right

2.4 Reward

For the reward and training setup, we follow the procedure of [Rahme et al., 2020b] with a few adjustments. To reduce the training computation, we reduce the maximum length of an episode from 5000 steps to 2000 steps, which translates to 20 seconds per episode. An episode is terminated prematurely if the robot hits the ground or exceeded a roll or pitch of 60 degrees. The reward function for each time step as described in the paper of D^2-GMBC is defined as:

$$r_t = \Delta d - 10(|r| + |p|) - 0.03 \sum |\omega| \tag{4}$$

The term Δd is the distance traveled in the forward direction in one step, and the penalty terms for r and p encourage the robot's body to remain perpendicular to the ground. The ω term further promotes a stable walk by punishing any angular motions of the body.

However, the code implementation as provided by the authors [Rahme et al., 2020a], uses d in the reward instead of Δd. This difference makes sense since the roll and pitch are many orders of magnitude larger than the traversed distance per step, Δd, which is in meters. The downside of this approach is that the reward per step for a given state becomes infeasible to determine with decent accuracy, since the reward at the end is much higher than at the beginning, and the time is not observed by the agent. To illustrate this problem, consider a speed of 0.2 m/s, the distance part of the reward can differ by 4.0 between the beginning and end state of an episode. This is in the same range as the roll and pitch reward. For the training of the ARS agent, his discrepancy is not a problem because, as the following sections explain, the policy is updated only by the total reward of an episode. Contrary, the SAC agent updates its policy based on the reward obtained at each step. For better comparison of the RL agents we train them with the same objective and modify the reward to be:

$$r_t = 100\Delta d - 10(|r| + |p|) - 0.03 \sum |\omega| \tag{5}$$

This scaling factor was chosen to put the distance reward in the same range as the orientation penalty, and resulted in the best convergence on both agents during early experiments.

2.5 Soft Actor Critic

An Actor-Critic algorithm is an off-policy method that computes both the policy and the value function over states. As mentioned, the policy defines a distribution over the actions for a given state. The value function $V(s)$ estimates the expected return for each state, under the current policy. Similarly, the Q-function estimates the expected return of each state-action pair and relates to the value function by $V(s) = \mathbb{E}_{a\sim\pi(s)} Q_\pi(s, a)$.

Typically dynamical programming is used for the Policy iteration algorithm to find the optimal policy by breaking the optimization process up into two steps. The first step is policy evaluation, where the value function is computed for the current policy. Then during policy improvement, the new value function is used to optimize the policy. In the Actor-Critic algorithm, these functions are updated simultaneously.

In this report, we apply the Soft Actor-Critic (SAC) algorithm as proposed by [Haarnoja et al., 2018a], where the critic is the Q-function. For SAC an extra constraint is introduced to the optimization of the value and policy function, whereby the entropy $\mathcal{H}$ is maximized. Entropy is a measurement of how uncertain the policy is to choose an action for a given state. This means that if the policy is deterministic, the entropy of the policy would be minimal. Maximizing the entropy, therefore, encourages exploration of the state space, which in turn reduces the risk that the policy gets stuck in a local optimum.

During each step of training, the replay buffer $\mathcal{D}$ stores the step tuple $(\mathbf{s}_t, \mathbf{a}_t, \mathbf{s}_{t+1}, \mathbf{r}_t)$. After enough steps have been stored, a batch of steps is sampled from the buffer to update the functions. A common problem with learning a Q-function is the influence of a positivity bias, where the model puts more emphasis on positive rewards than negative rewards. To minimize the positivity bias and improve learning speed, double Q-learning is applied, a method where two Q-functions are trained in parallel, parameterized by μ_1, μ_2. More specifically we apply the double Q-learning procedure as proposed in [Fujimoto et al., 2018], by using the minimum value of the two Q-functions for the value gradient. Minimizing the loss:

$$J_Q(\mu_i) = \mathbb{E}_{(\mathbf{s}_t, \mathbf{a}_t, \mathbf{s}_{t+1}, \mathbf{r}_t)\sim\mathcal{D}} \left[\left(Q_{\mu_i}(\mathbf{s}_t, \mathbf{a}_t) - (\mathbf{r}_t + \gamma V_{\mu_1,\mu_2}(\mathbf{s}_{t+1})) \right)^2 \right] \tag{6}$$

This loss minimizes the squared error between the current state-action value and what it should be according to the minibatch, namely the real reward plus the state value of the next state, according to the current policy. The value function in equation 6 is implicitly defined by the policy and the Q-function as:

$$V_{\mu_1,\mu_2}(\mathbf{s}_t) = \mathbb{E}_{\mathbf{a}_t \sim \pi_\theta}\left[\min_{i\in\{1,2\}} Q_{\mu_i}\left(\mathbf{s}_t, \mathbf{a}_t\right) - \alpha \log \pi_\theta\left(\mathbf{a}_t \mid \mathbf{s}_t\right)\right] \tag{7}$$

The policy is learned by minimizing the Kullback-Leibner distance between the action distribution and the state-action distribution, for a sampled minibatch of states. This policy loss as defined below is the same as the policy loss of the original SAC [Haarnoja et al., 2018b] but uses the explicit temperature parameter α, which relates to the entropy.

$$J_\pi(\theta) = \mathbb{E}_{\mathbf{s}_t \sim \mathcal{D}, \mathbf{a}_t \sim \pi_\theta}\left[\alpha \log \pi_\theta\left(\mathbf{a}_t \mid \mathbf{s}_t\right) - \min_{i\in\{1,2\}} Q_{\mu_i}\left(\mathbf{s}_t, \mathbf{a}_t\right)\right] \tag{8}$$

Because the action batch is sampled from the policy, the loss could not be backpropagated to update θ, therefore the action is sampled using the reparameterization trick, where the stochasticity of the sampling arises from an external sampled noise variable. Given this noise variable, the action outputted by the policy is obtained in a deterministic manner. Lastly, the temperature parameter is updated by minimizing:

$$J(\alpha) = \mathbb{E}_{\mathbf{s}_t \sim \mathcal{D}, \mathbf{a}_t \sim \pi_\theta}\left[-\alpha \log \pi_\theta\left(\mathbf{a}_t \mid \mathbf{s}_t\right) - \alpha \mathcal{H}\right] \tag{9}$$

A summary of the algorithm automatic entropy adjusting SAC algorithm is shown in Algorithm 1

Algorithm 1 Soft Actor-Critic with Automatic Entropy Adjustment from [Haarnoja et al., 2018a]

for each iteration **do**
  for each environment step **do**
    $a_t \sim \pi_\mu(a_t|s_t)$
    $s_{t+1} \sim p(s_{t+1}|s_t, a_t)$
    $D \leftarrow D \cup \{(s_t, a_t, r(s_t, a_t), s_{t+1})\}$
  end for
  for each gradient step **do**
    $\mu_i \leftarrow \mu_i - \lambda \hat{\nabla}_\mu J_Q\left(\mu_i\right)$ for $i \in \{1, 2\}$
    $\theta \leftarrow \theta - \lambda \hat{\nabla}_\theta J_\pi(\theta)$
    $\alpha \leftarrow \alpha - \lambda \nabla_\alpha J(\alpha)$
  end for
  $\bar{\mu}_i \leftarrow \tau \mu_i + (1-\tau)\bar{\mu}_i$ for $i \in \{1, 2\}$
end for

2.6 Augmented Random Search

Where standard RL algorithms can never know which set of parameters would be better in a stochastic environment, ARS applies different rollouts in parallel, where the sampled environment variables remain the same. In simulation training, this means that for each episode, 16 rollouts are performed under the same sampled parameters for the dynamics and the domain. Each rollout thus contains a slightly different permutation of the previous best policy. The policy function of ARS consists of a single layer Neural Network θ, that maps from the observation space to the action space. Each rollout i uses its own policy $\bar{\theta}_i = \theta + \Delta\theta_i$, by sampling a set of residual parameters $\Delta\theta_i$ from a normal distribution with mean 0 and standard deviation 0.05. The best policy then updates the current policy with a learning rate of 0.03. The implementation details of ARS remain the same to [Rahme et al., 2020b]. From the provided code implementation[4] we gather that an additional action filter was used for the implementation of ARS. To illustrate this filter, let new_action be the action as determined by the policy in the current state, and

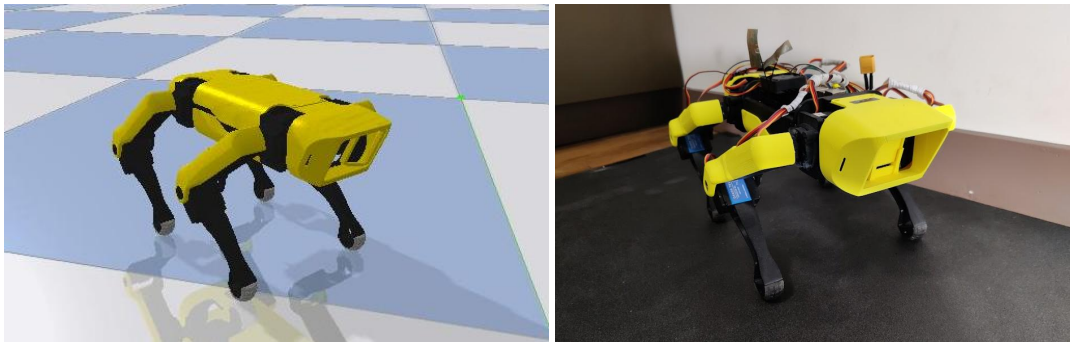
[4] https://github.com/OpenQuadruped/spot_mini_mini

old_action be the action applied by the agent in the previous state. The action that is then applied in the current state is:

$$action = \alpha * old\_action + (1 - \alpha) * new\_action \tag{10}$$

This filter reduces the risk of abrupt motions during training, but offers the agent no way to get a good set of actions at the start of an episode. The agent is therefore kept fixed for the first 20 steps of each episode so that the action values have time to get to a decent baseline. To improve the quality of the comparison between the training procedures of ARS and SAC, we wanted to keep the observation space, the environment, and the action space the same for the ARS and SAC models. However, early experiments showed that training SAC using the alpha filter prevented learning. An explanation for this learning failure is that the alpha filter breaks the Markov Property, which states that each next state is only dependent on the current state, action, and observation, and independent of other previous states and actions. But the alpha filter makes the model dependent on the older actions as well. Conversely, training ARS without the alpha filter also resulted in failed learning. Therefore, we keep the alpha filter for the ARS implementation and remove it for the SAC implementation.

2.7 The Robot



(a) Simulation (b) Real robot

Figure 3: The constructed SpotMicroAI in Pybullet simulation and the physical robot

2.7.1 The physical robot

The body of the robot is 3D printed from Polylactic Acid (PLA) material except for the feet, which are made of rubber for better shock absorption. To handle the weight of the robot all the servos have a maximum torque of 20kg/cm. Although plenty of schematics of the Spotmicro are available online, there is no good solution that merges the Jetson Nano with the actuators and IMU sensor. Through careful examination, a new schematic is created which is displayed in Figure 4. In early experiments a Jetson Nano broke and to prevent further problems with the power supply of the servos and the Jetson, they are given their separate power supply. The actuators are powered by a battery that supplies 5v at a max of 20A, and the Jetson is powered through an adapter at the Barrel Jack with 10W. During experiments, the cable is lifted loosely close to the body to minimize weight or drag hindrances caused by the cable. Lastly, the top back component is left out because the Jetson extended too high, and to compensate for the extra weight of the Jetson.

2.7.2 Simulation specifics

All simulation training is performed on the physics simulator Pybullet [Coumans and Bai]. The 3D model of the Spotmicro in simulation is transferred from the mesh used to 3D print the components and therefore allows for faithful reconstruction of the real robot. Additional weight is placed inside the robot to mimic the weight influence of the Jetson Nano and the battery. The inertia values of each component are left at the default of (100, 100, 100). Although it is known that it is a very crude approximation of the real inertia, finding the correct values is labor intensive, and by the different randomization factors of the

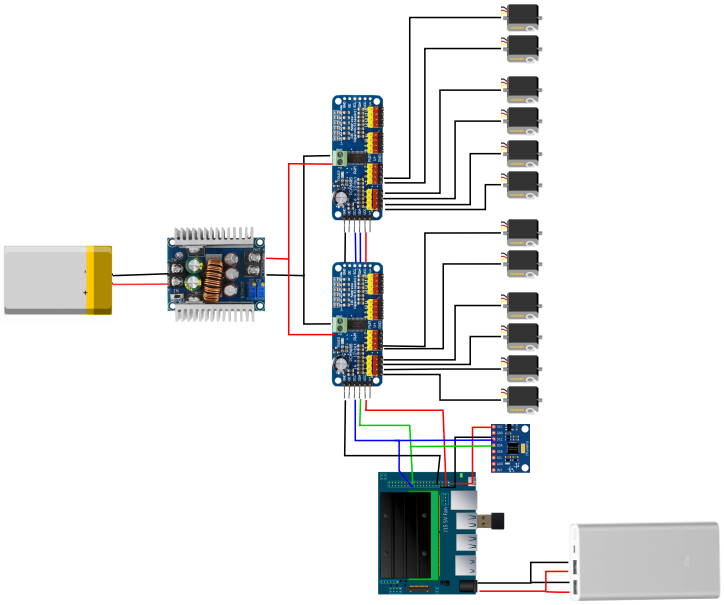


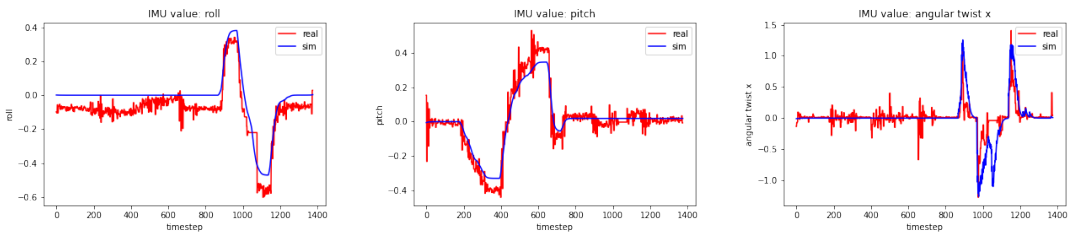
Figure 4: Schematic of the electronics components of the SpotMicro

training, we hope to reduce the side effects. The motor parameters are transferred from D^2-GMBC, and the height is modified to keep the body 18cm above the ground.

3 Results & Discussion

3.1 IMU comparison

Before the quality of the RL agents on the real robot can be inspected we first compare the IMU data between the simulation and the real robot. To achieve this comparison, we use IK to make the robot apply a roll in both directions followed by a pitch in both directions. In Figure 5 three of the eight measurements are displayed. For a full comparison of the measurements see Appendix A. All displayed values were as returned by the IMU, except for the linear acceleration, which was reduced by a factor of 10 to match the values in the simulation. As seen from the Figures, the roll and pitch measurements are similar to those in the simulation. The angular twist also follows the same values as the simulation, but already contains a lot more noise, especially in the z-direction. Lastly, the linear acceleration looks much more chaotic in real life than in simulation but importantly has spikes at the same points in the graph which suggests it still contains useful information. Even though the angular twist and linear acceleration contain more noise on the real sensor than in the simulation, the results in the simulation also show erratic behavior, we thus expect the agents to put more weight on the values of the roll and pitch. From these results, we conclude that the sensor data provided by the IMU is faithful enough for the sim-to-real transfer of the RL agents.



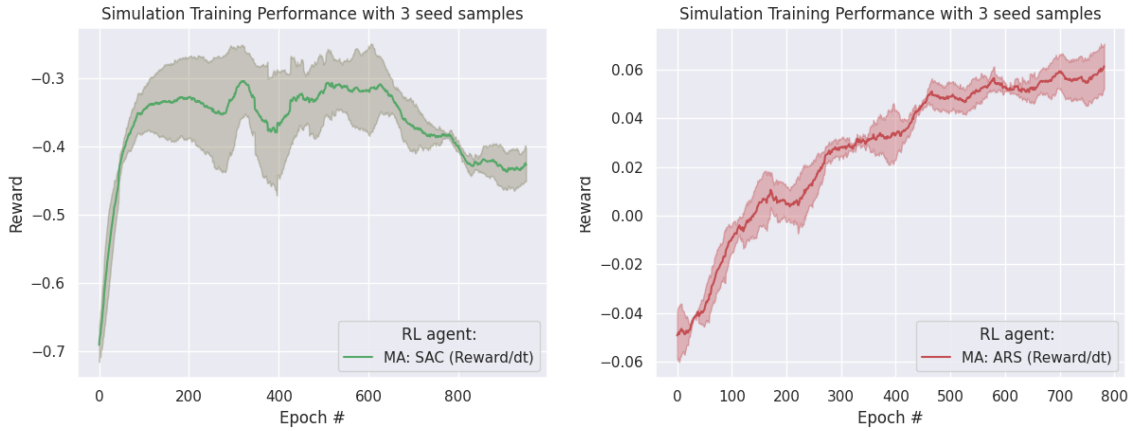
(a) Roll (b) Pitch (c) Angular twist: x-direction

Figure 5: IMU results of the same movement in simulation and on the real robot. In red are the results of the real robot and in blue are the results of the simulation. The performed movement consisted of a negative and positive pitch movement, followed by a positive and negative roll movement

3.2 Simulation

The training performance of the SAC and ARS agents in different environments is shown in Figure 6. Experiments of both agents were performed over the same three random seeds. The mean of these experiments is represented by the dark line and the lighter area represents the standard deviation. Similar to the figure of the D^2-GMBC paper, we publish the moving average window over 50 episodes. Since the ARS agent uses an action filter and remains stationary for the first 20 time steps there is a training discrepancy that makes direct comparison of the rewards unattainable. Therefore the episode reward as depicted in Figure 6 starts when the agent commences walking and stops T-20s later. So for ARS 20s-5000s, and for SAC from 0s-4980s. Note that this is only for illustrative purposes, the reward used in learning for ARS and SAC was based on the entire run and not this specific selection.

From the learning curves in Figure 6 we see that although the starting reward of SAC is much lower, it learns much faster than ARS. This is in line with our expectations since ARS only updates its policy once per episode and SAC after each episode step after the replay buffer is full enough. However, we see that ARS achieves a much higher reward than SAC and that the reward of ARS always increases over time. Both this non-decreasing reward and the much higher reward of ARS training are a result of the multiple rollouts. For each of the episodes, one of the 16 rollouts will likely lead to a policy with a higher reward. To estimate the quality of these networks we need to investigate if ARS retains its advantage beyond the benefit of its privileged information through the rollouts. Therefore the next section evaluates both models on our own built version of the Spotmicro.



(a) SAC (b) ARS

Figure 6: Training curves in simulation for the two agents using D^2 randomization. Left in green the agent trained with SAC, right in red is the agent trained with ARS.

3.3 Sim-to-real transfer

To inspect the efficacy of the sim-to-real transfer of the different models we replicate one of the experiments of [Rahme et al., 2020b] and compare how fast the robot walks a track of one meter. Because the legs of the real robot broke off on multiple occasions during early experiments, the track consists of a slightly flexible thin mat with a height of 0.5 cm, to reduce the impact when the legs touch the floor. The robot is tasked to walk this path ten times for each RL agent and for the baseline model which uses the fixed gait. Since the used mat is of limited width, the experiment was paused when the robot walked off the mat and the robot was moved to the middle of the track whilst maintaining the same distance from the start position. Table 1 summarizes the results of the experiments. For each agent, the mean and standard deviation of the results are shown. For each RL agent, one run was considered failed because the robot became stuck in continual erratic behavior that prevented it to go forward.

From table 1 we notice that neither of the agents was able to improve the forward walking speed above that of the baseline. When looking at the standard deviation we do note that the ARS agent has a more

	Mean	Standard deviation	Failed Runs
Baseline	15.20	2.56	0
ARS	15.33	0.94	1
SAC	21.22	1.47	1

Table 1: The results of real-world experiments of traversing one meter. The columns represent the mean and standard deviation of the time it took each agent.

predictable walking speed, as indicated by the low standard deviation. The SAC agent is significantly slower than the other two and has an average walking speed of 21 seconds per meter compared to the 15 seconds per meter of the other two models. However, the walking speed was only one part of the reward signal, so to evaluate the walking stability we need to inspect the quality of the walk itself.

3.3.1 Video analysis

To further analyze the different implementations we supply three short representative clips of the robot walking the track, see video[5]. Even though the robot parameters are tuned to optimize the fixed bezier gait without an agent, this baseline implementation still demonstrates a suboptimal walk. Even after meticulous calibration and reconfiguration of the servos, the robot falls through its hind legs on multiple occasions, likely due to the larger weight of the body arising from the Jetson Nano and the Battery. The robot still seems to achieve forward locomotion but the video material suggests the baseline gait can still improve much.

The ARS video shows adaptive behavior when the robot threatens to fall backward or forward. However, in the video it is also visible that the agent sometimes overcompensates, causing the body to swing from left to right or from front to back. Therefore we hypothesize that the reduced traversal time likely arises from the falling prevention behavior.

The SAC agent displays even more complex falling prevention behavior. In the video, this is visible when the roll or pitch becomes too high, causing the agent to stand still for a short term. Thus the agent can not only modify one or two legs based on the body angles but when the IMU values diverge significantly enough, the agent even seems to halt all movement. However, from the start of the SAC clip, we see this falling prevention can hinder the gait significantly since it has troubles with starting the gait. Yet, after the agent has achieved a stable gait, the SAC agent demonstrates significant adaptive behavior.

4 Conclusion

In this report, we extended the work of [Rahme et al., 2020b] on open-loop Bezier gait modulation and evaluated how deeper RL methods hold up against their ARS implementation. During simulation training of the agents, ARS seemed to achieve a much higher reward than our SAC implementation. After further investigation, this advantage is suspected to be mainly due to the 16 parallel roll-outs of the ARS agent which makes it hard to compare training efficiency. From the experiments on the real robot we found that although neither of the agents was faster than the baseline gait, the RL agents did display adaptive falling prevention behavior. This behavior was most visible in the SAC agent, but also seemed to be the cause of the reduced traversal speed. A potential solution for this behavior is to increase the traversal speed in the reward so the agent is less concerned with minor deviations in the body orientation.

References

Erwin Coumans and Yunfei Bai. Pybullet, a python module for physics simulation in robotics, games and machine learning., 2016-2017. *URL http://pybullet. org*.

Scott Fujimoto, Herke Hoof, and David Meger. Addressing function approximation error in actor-critic methods. In *International conference on machine learning*, pages 1587–1596. PMLR, 2018.

[5] https://youtu.be/Ji8rXjD60mU

Tuomas Haarnoja, Sehoon Ha, Aurick Zhou, Jie Tan, George Tucker, and Sergey Levine. Learning to walk via deep reinforcement learning. *arXiv preprint arXiv:1812.11103*, 2018a.

Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International conference on machine learning*, pages 1861–1870. PMLR, 2018b.

Danijar Hafner, James Davidson, and Vincent Vanhoucke. Tensorflow agents: Efficient batched reinforcement learning in tensorflow. *arXiv preprint arXiv:1709.02878*, 2017.

Marco Hutter, Christian Gehring, Dominic Jud, Andreas Lauber, C Dario Bellicoso, Vassilios Tsounis, Jemin Hwangbo, Karen Bodie, Peter Fankhauser, Michael Bloesch, et al. Anymal-a highly mobile and dynamic quadrupedal robot. In *2016 IEEE/RSJ international conference on intelligent robots and systems (IROS)*, pages 38–44. IEEE, 2016.

Jemin Hwangbo, Joonho Lee, Alexey Dosovitskiy, Dario Bellicoso, Vassilios Tsounis, Vladlen Koltun, and Marco Hutter. Learning agile and dynamic motor skills for legged robots. *Science Robotics*, 4(26): eaau5872, 2019.

Dong Jin Hyun, Sangok Seok, Jongwoo Lee, and Sangbae Kim. High speed trot-running: Implementation of a hierarchical controller using proprioceptive impedance control on the mit cheetah. *The International Journal of Robotics Research*, 33(11):1417–1445, 2014.

Atil Iscen, Ken Caluwaerts, Jie Tan, Tingnan Zhang, Erwin Coumans, Vikas Sindhwani, and Vincent Vanhoucke. Policies modulating trajectory generators. pages 916–926, 2018.

Joonho Lee, Jemin Hwangbo, Lorenz Wellhausen, Vladlen Koltun, and Marco Hutter. Learning quadrupedal locomotion over challenging terrain. *Science robotics*, 5(47):eabc5986, 2020.

Xue Bin Peng, Marcin Andrychowicz, Wojciech Zaremba, and Pieter Abbeel. Sim-to-real transfer of robotic control with dynamics randomization. In *2018 IEEE international conference on robotics and automation (ICRA)*, pages 3803–3810. IEEE, 2018.

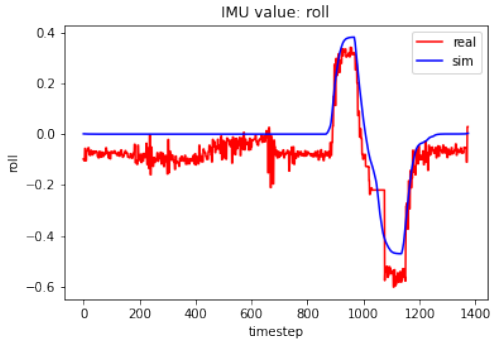
Maurice Rahme, Ian Abraham, Matthew Elwin, and Todd Murphey. Spotminimini: Pybullet gym environment for gait modulation with bezier curves, 2020a. URL https://github.com/moribots/spot_mini_mini.

Maurice Rahme, Ian Abraham, Matthew L Elwin, and Todd D Murphey. Dynamics and domain randomized gait modulation with bezier curves for sim-to-real legged locomotion. *arXiv preprint arXiv:2010.12070*, 2020b.

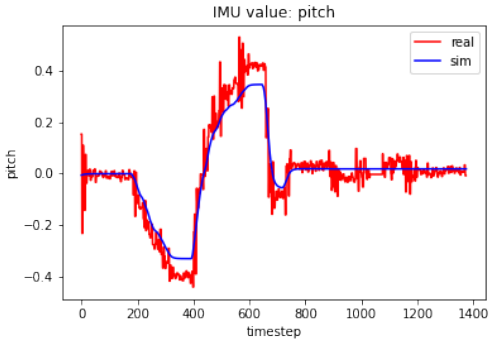
Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

Jie Tan, Tingnan Zhang, Erwin Coumans, Atil Iscen, Yunfei Bai, Danijar Hafner, Steven Bohez, and Vincent Vanhoucke. Sim-to-real: Learning agile locomotion for quadruped robots. *arXiv preprint arXiv:1804.10332*, 2018.

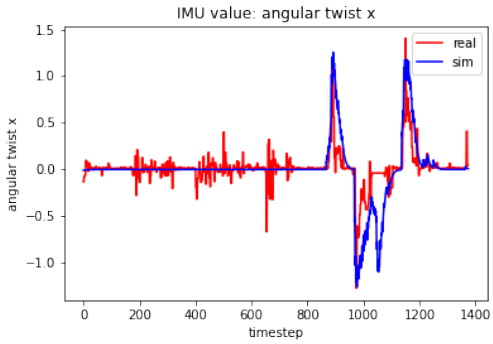
A Appendix IMU values



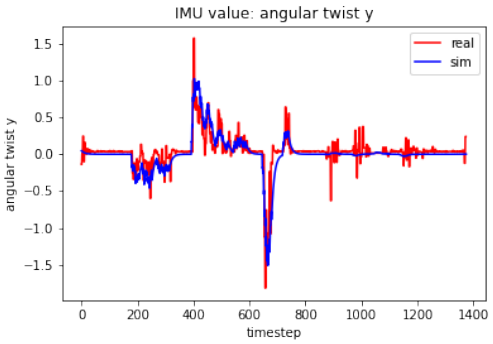
(a) Roll



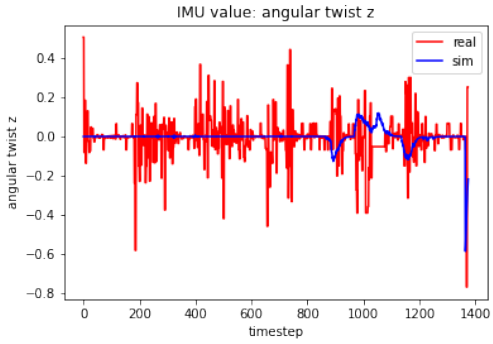
(b) Pitch



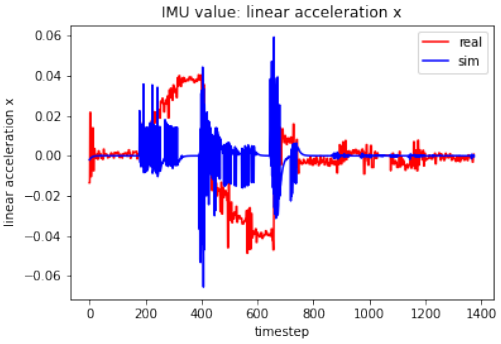
(c) Angular twist: x-direction



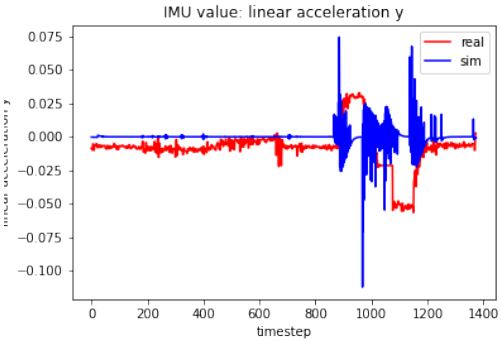
(d) Angular twist: y-direction



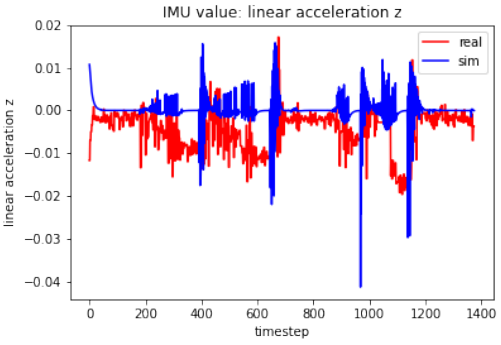
(e) Angular twist: z-direction



(f) Linear Acceleration: x-direction



(g) Linear Acceleration: y-direction



(h) Linear Acceleration: z-direction

Figure 7: IMU results of the same movement in simulation and on the real robot. In red are the results of the real robot and in blue are the results of the simulation. The performed movement consisted of a negative and positive pitch movement, followed by a positive and negative roll movement