Reinforcement Learning for Quadrupedal Locomotion

Michael Freeman

April 2021

Abstract

Creating a controller for legged robots is a notoriously difficult problem requiring in-depth knowledge of the robot and its mechanisms. For this reason, it may be easier to use machine learning to learn a controller for the system. Such learned controllers may also be more optimal than those programmed by their designers. A common problem faced with such techniques is a disparity between the performance of a learned controller in simulation and in reality, known as the Reality Gap. Recently efficient algorithms have been developed to avoid the reality gap by learning in the real world rather than in simulation. Soft Actor-Critic (SAC), one such sample efficient algorithm is implemented and applied to the problem of quadrupedal locomotion in a simulation. This method is tested with varying weightings of the reward function of the environment, achieving varying results.

I certify that all material in this dissertation which is not my own work has been identified. Michael Freeman

Contents

1 Introduction

Robots with that can walk are becoming increasingly popular do to the many advantages they have over their wheeled counterparts. These advantages include environmental versatility, i.e. a robot with legs can handle rough terrain far better than one with wheels. For example these robots could be used to aid mountain rescue or to replace or go in tandem with the Mars rovers to explore the less accessible regions of the planet. However, programming a controller for such robots is a far more difficult problem than it is for wheeled robots due to the extra dimensions that legs add. This would require deep knowledge of the workings of the robot and the planning and development of a walking gait. This complexity makes it an ideal candidate to be made into a reinforcement learning problem.

Reinforcement learning is a category of machine learning, focussed on an agent learning the best possible solution to a problem independently from experience. The idea is that given a set $\mathcal{S}$ of possible states and a set $\mathcal{A}$ of possible actions, we define a reward function $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$. We then try to find a policy, $\pi : \mathcal{S} \to \mathcal{A}$, that maximises the reward received over time. That is find:

$$\pi^* = \arg\max_{\pi} \sum_{t} \mathbb{E}_{s_t, a_t \sim \pi} \gamma^t r(s_t, a_t) \tag{1}$$

where $\gamma \in (0, 1]$ is a discount factor that helps this sum to converge should it be infinite.

A problem, however, with using a reinforcement learning approach is that traditional approaches require large amounts of data to be able to learn even simple, low dimensional tasks, such as playing early video-games and Go. The problem of teaching a robot to walk is high dimensional, however, meaning that even larger amounts of data are needed. In recent years as computing power has become more readily available this has become less of a problem for the lower dimensional tasks, however still remains for those with higher observation and action space dimensions. To solve this problem sample efficient algorithms have begun to be developed, that is algorithms that learn as much as possible from as little data as possible. I will be implementing one such algorithm to apply it to the problem of a quadrupedal robot walking .

2 Literature Review

2.1 Conventional Methods and the Reality Gap

The conventional method for reinforcement learning would be to build a simulation of the real world problem, run your learning algorithm on this simulation and then apply the learned behaviour to the real world problem. The problem with this solution, however, is that no simulation will perfectly describe reality. These discrepancies then build up and result in learned behaviour that performs well in simulation but once applied to the real world perform poorly. This phenomenon, called the reality gap, is described by Tan *et al.* [1] who also propose a solution.

This solution involves creating a highly accurate simulation by de-constructing the robot, analysing each of its parts and using this to create an accurate simulated model. The second part of their solution involves leaning a robust walking gait, that is a gait that performs well in the presence of model error. The method that they use to make their model robust is to add perturbations of random direction and magnitude to simulation as it is training to model the error it may face when transferred to real life.

Tan *et al.* show that this method works and can achieve real word results close to that of the simulation, however, requires a large amount of work to apply it to a new robot. The required overhead of developing a highly accurate simulation almost defeats the object of learning a gait rather than hard coding it.

2.2 Real World Learning and Soft Actor-Critic

Another method to minimise the effect of the reality gap is to avoid it altogether by learning in real life. The problem with this, however, is that many traditional reinforcement learning algorithms require large amounts of data to learn well. For example it took 4.8 million games to learn how to play Go, a problem with a relatively low dimensionality [2]. It would clearly be infeasible to apply these traditional methods to learning in real life on a high dimensional problem like legged locomotion. For this reason algorithms that can learn with as little data as possible have been developed.

One such algorithm is Soft Actor-Critic (SAC) [3]. The idea behind SAC is to maximise entropy while learning. This allows for the algorithm to sample widely from the action-space as it trains to explore as much of it as possible, while also reinforcing good behaviours. This aims to prevent the algorithm getting stuck at a local maximum and aims to find the global maximum. Haarnoja *et al.* find that the entropy temperature hyper-parameter is sensitive to change [4], however tuning this can be combined with the main learning as a gradient descent problem.

2.3 Project Specification

I implement the Soft Actor-Critic algorithm and apply this to the problem of teaching a quadrupedal robot to walk. In the main Literature review submitted previously I had stated that I would be developing a method for the robot to walk to a specified point in the environment. However I decide that I will no longer develop this part of the project. This is in part due to the lack of literature on the subject of developing an autonomous controller and that learning a walking gait is a hard enough problem as it is.

3 Design

3.1 Technologies

Throughout the project I use Python, with Pybullet [5] and Pytorch packages. I choose Python due to its popularity in machine learning and the vast number of

packages that support it. While it is perhaps not as fast as languages such as C or C++, Python allows for neat, readable code to be prototyped and written quickly.

I use Pytorch over Tensorflow or Keras, for evaluation of Neural Networks, because of its syntax being "pythonic" and friendlier to use. It shares many of the same features as Tensorflow and Keras so the choice is mainly that of preference. Both Tensorflow and Pytorch allow for CUDA GPU acceleration, however Tensorflow has better support for TPU acceleration which allows even faster computation. This shouldn't be a problem though as I am unlikely to need the use of a TPU.

There were many different simulators that I could have used to simulate the robot. I use Pybullet for multiple reasons. Since it is a python package this allows for simple integration with any learning code that I write. This also prevents the need for an extra program. Pybullet is also a popular choice for reinforcement learning tasks on robots and as such has a model and OpenAI Gym [6] environment for the minitaur robot used in [7] and [1]. The minitaur simulation is the highly accurate simulation developed by Tan *et al.* to minimise the reality gap [1].

3.2 Soft Actor-Critic

The Soft Actor-Critic (SAC) algorithm has two generations which implement the same idea differently. The general idea of reinforcement leaning is the maximise the reward over time as seen in equation 1, however in SAC we also want to maximise entropy so we modify this to a more general from [8]:

$$\sum_t \mathbb{E}_{s_t,a_t\sim\pi} r(s_t,a_t) + \alpha\mathcal{H}\pi(\cdot|s_t) \tag{2}$$

where α is the entropy temperature and $\mathcal{H}$ is the target entropy. The implementation of the entropy term is where the generations of the algorithm differ. In both generations we need a policy function π parametrised by ϕ which is a Gaussian network and soft Q functions parametrised θ_i. In the first generation of SAC entropy is modelled with a value function V parametrised by ψ. Our overall objective is to minimise the loss function of π defined as:

$$J_\pi(\phi) = \mathbb{E}_{a_t,s_t\sim\mathcal{D}}[\log\pi_\phi(a_t|s_t) - Q_\theta(s_t,a_t)] \tag{3}$$

Here $\mathcal{D}$ is a replay memory to hold information of the last N steps in the learning process, the details of which are explained in section 3.3.1. In training we also wish to optimise our soft Q and Value functions. We use the Mean Squared Error (MSE) loss for these:

$$J_V(\psi) = \mathbb{E}_{s_t\sim\mathcal{D}}[\frac{1}{2}(V_\psi(s_t) - \mathbb{E}_{a_t\sim\pi_\phi}[Q_\theta(s_t,a_t) - \log\pi_\phi(a_t|s_t)])^2] \tag{4}$$

$$J_Q(\theta) = \mathbb{E}_{(s_t,a_t)\sim\mathcal{D}}[\frac{1}{2}(Q_\theta(s_t,a_t) - \hat{Q}(s_t,a_t))^2] \tag{5}$$

where

$$\hat{Q}(s_t,a_t) = r(s_t,a_t) + \gamma\mathbb{E}_{s_{t+1}\sim p}[V_{\bar{\psi}}(s_{t+1})] \tag{6}$$

The extra parameter for the V function, $\bar{\psi}$, in equation 6, is the target V function, which is added to add stability to learning. If the normal parameter ψ was used instead there would be instability as the loss function for ψ depends on θ and the loss for θ depends on ψ.

Using these loss functions we can now define the SAC algorithm, Algorithm 1. The algorithm uses two separate Q functions parametrised separately by θ_1 and θ_2, each of which is optimised separately. This is done to speed up learning on high dimensional tasks such as quadrupedal locomotion. Whenever a Q value is referenced in a loss function we therefore use the minimum of the two functions. The parameters λ_V, λ_Q and λ_π are the learning rates for the value, Q and policy functions respectively and τ is the smoothing constant for the target value function. Line 6 in Algorithm 1 gets the next state, s_{t+1}, after applying the action a_t to the environment at step s_t.

Algorithm 1: SAC1

Input: ϕ, ψ, θ_1, θ_2, λ_V, λ_Q, λ_π, τ

1 $\bar{\psi} \leftarrow \psi$;
2 $\mathcal{D} \leftarrow \emptyset$;
3 **foreach** *iteration* **do**
4 **foreach** *environment step* **do**
5 $a_t \sim \pi_\phi(a_t|s_t)$;
6 $s_{t+1} \sim p(s_{t+1}|s_t, a_t)$;
7 $\mathcal{D} \leftarrow \mathcal{D} \cup \{(s_t, a_t, r(s_t, a_t), s_{t+1})\}$;
8 **end**
9 **foreach** *update step* **do**
10 $\psi \leftarrow \psi - \lambda_V \hat{\nabla}_\psi J_V(\psi)$;
11 $\theta_i \leftarrow \theta_i - \lambda_Q \hat{\nabla}_{\theta_i} J_Q(\theta_i)$ for $i \in \{1, 2\}$;
12 $\phi \leftarrow \phi - \lambda_\pi \hat{\nabla}_\phi J_\pi(\phi)$;
13 $\bar{\psi} \leftarrow \tau\psi + (1 - \tau)\bar{\psi}$;
14 **end**
15 **end**

The second generation of the algorithm removes the need for a separate value function to model entropy, instead approximating the value function as a function of the policy function, Q function and an entropy temperature, α. We have:

$$V(s_t) = \mathbb{E}_{a_t \sim \pi}[Q(s_t, a_t) - \alpha \log \pi(a_t|s_t)] \tag{7}$$

This slightly alters the loss functions defined earlier. We now have:

$$J_\pi(\phi) = \mathbb{E}_{a_t, s_t \sim \mathcal{D}}[\alpha \log \pi_\phi(a_t|s_t) - Q_\theta(s_t, a_t)] \tag{8}$$

$$J_Q(\theta) = \mathbb{E}_{(s_t, a_t) \sim \mathcal{D}}[\frac{1}{2}(Q_\theta(s_t, a_t) - (r(s_t, a_t) + \gamma \mathbb{E}_{s_{t+1} \sim p}[V_{\bar{\theta}}(s_{t+1})]))^2] \tag{9}$$

Note that in equation 9 we introduce a new parameter $\bar{\theta}$. This is introduced to stabilise the learning for the same reasons as with $\bar{\psi}$ previously.

Since we also wish to optimise the entropy temperature parameter, α, we also need a loss function for this:

$$J(\alpha) = \mathbb{E}_{a_t \sim \pi}[-\alpha \log \pi(a_t|s_t) - \alpha\mathcal{H}] \tag{10}$$

where $\mathcal{H}$ is the minimum target entropy.

We can now use these to define the SAC2 algorithm, Algorithm 2. This is very similar to Algorithm 1, but with slightly different parameters being trained. As before the λ terms are the learning rates for their respective parameters and τ is a smoothing constant, but this time for the target Q parameters.
I implement both of these algorithms to compare the performance of both.

Algorithm 2: SAC2

Input: ϕ, θ_1, θ_2, α, λ_α, λ_Q, λ_π, τ

$\bar{\theta_1} \leftarrow \theta_1, \bar{\theta_2} \leftarrow \theta_2$;
$\mathcal{D} \leftarrow \emptyset$;
foreach *iteration* **do**
  foreach *environment step* **do**
    $a_t \sim \pi_\phi(a_t|s_t)$;
    $s_{t+1} \sim p(s_{t+1}|s_t, a_t)$;
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{(s_t, a_t, r(s_t, a_t), s_{t+1})\}$;
  end
  foreach *update step* **do**
    $\theta_i \leftarrow \theta_i - \lambda_Q \hat{\nabla}_{\theta_i} J_Q(\theta_i)$ for $i \in \{1, 2\}$;
    $\phi \leftarrow \phi - \lambda_\pi \hat{\nabla}_\phi J_\pi(\phi)$;
    $\alpha \leftarrow \alpha - \lambda_\alpha \hat{\nabla}_\alpha J(\alpha)$;
    $\bar{\theta}_i \leftarrow \tau\theta_i + (1 - \tau)\bar{\theta}_i$ for $i \in \{1, 2\}$;
  end
end

3.3 Representing SAC

3.3.1 Replay Memory

In calculating the loss functions we sample states and actions that have occurred previously. To do this we need to create a replay memory. While this could just be a list that records and keeps information on all steps that have happened, this would be infeasible storage-wise for more complex tasks that require long training periods. Instead this could be implemented as a finite list, saving information only on the last N steps. An advantage of doing it this way is that as the algorithm trains the policy function to be better we are sampling from better transitions, as opposed to all including the random data fed in at the beginning.

This must consist of a list, a capacity and a memory position. As a new state transition is pushed to the memory it should be put in the list at the memory position, which increments modulo the capacity with each push. We also need a sample function to produce a random sample of the replay memory of a specified size.

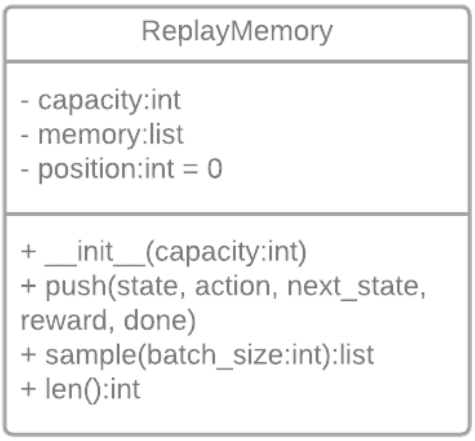


Figure 1: UML diagram for Replay Memory

3.3.2 Neural Networks

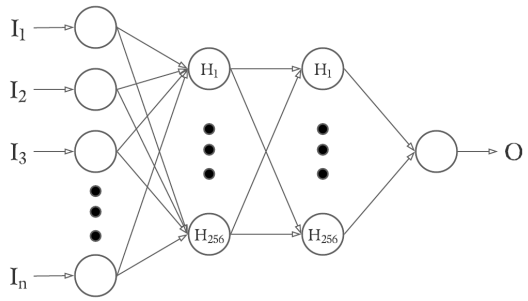
The policy, value and Q functions used in SAC can be represented as artificial neural networks, where the parameters ϕ, ψ and θ are the network weights. The update steps for the parameters in Algorithms 1 and 2 then translate to back-propagation. Haarnoja *et al.* [4], suggest that all networks used should have 2 hidden layers each with 256 units. They also suggest that the Rectified Linear Unit (ReLU) activation function and Adam [9] optimiser are used, where:

$$relu(x) = \max(0, x) \tag{11}$$

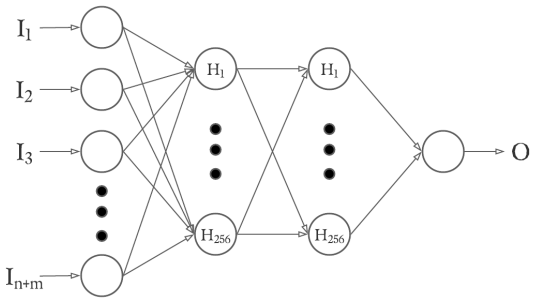
The Value function takes states as input and produces a value of this state, hence should have $n := \dim \mathcal{S}$ input units and a single output. Likewise the Soft Q network takes states and actions and returns a value so should have $n + m$ input units and a single output, where $m := \dim \mathcal{A}$. The policy network has a slightly different structure to the value and soft Q networks. The policy function takes states and returns a Gaussian probability distribution for the actions given the current state. It returns the means and standard deviations for each dimension of the action space. Therefore it takes n inputs and produces $2m$ outputs. Where we only need to be able to feed forward and back-propagate in the value and Q networks, our policy network also needs a function to return an action and its probability given a state. The structure of these networks can be seen in figure 2.

3.4 Reward Function

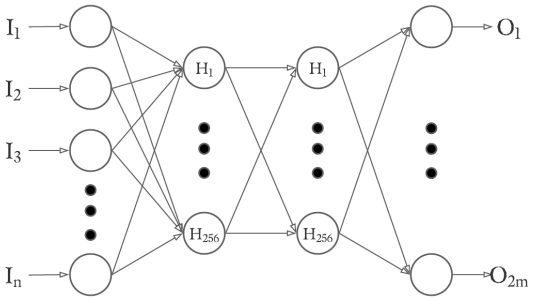
An important part of reinforcement learning is the reward function. This is what allows us to encourage good actions and discourage bad actions. The reward function I use for teaching the minitaur robot to walk is built into the pybullet environment [5]. This is a weighted sum of the forward movement f, sideways movement s, sideways rotation or shake r and the energy used e.



(a) Value network



(b) Q network



(c) Policy network

Figure 2: Neural network structures

$$\text{reward} = \omega_f f - \omega_s s - \omega_r r - \omega_e e \tag{12}$$

where each ω is a weight. Here energy is calculated as the dot product of the motor velocities and the motor torques. By default the weights are set as $\omega_f = 1, \omega_s = 0, \omega_r = 0$ and $\omega_e = 0.005$. I will need to tune these weight parameters to find a best reward function that promotes walking in a straight line.

For all tests ω_f is kept constant at 1 and the other weights are varied. This is done as the objective is to maximise the forward motion and all of the other weights punish for deviating from this objective. Energy consumption isn't a largely important factor in walking in a simulation, however may be when learning on a real robot to prevent damage and preserve battery. Therefore I don't explore this parameter much to reduce the number of parameters that need tuning. I do, however, do all experiments with the default $\omega_e = 0.005$ and $\omega_e = 0$. We may see that policies trained with a reward that takes energy into account has a more natural gait, where on trained with no energy penalty has a more chaotic gait.

For each of ω_s and ω_r I will test the learning at $\frac{1}{8}$ increments from 0 to 1, keeping the other constant at 0, then test combinations of these to find a best reward function. Should I find that all of these perform poorly I will investigate weights closer to the default of ω_e as it may be that the punishments are too severe. As the policy trains I record the reward of each walk. Due to the fact that the algorithm encourages exploration of the action space we will get points during training where the reward suddenly dips or spikes. For this reason I will also record the rolling average of the training to better show the trend of the learning.

I won't be able to directly compare the returned reward of runs with different reward functions, since a run with no penalty reward function will get a higher reward then a run with some penalty for doing the same thing. However, these will give a good idea on the performance of each policy. To compare, however, the final policy for each reward function can be tested on an environment with no penalties on the reward function.

4 Development

4.1 Neural Networks

In developing the project it makes most sense to start with developing the neural networks. Using pytorch it is fairly straightforward to quickly design the value and soft Q networks as described in section 3.3.2. The structure of the network is described in the constructor by defining linear layers with the respective input and output sizes. The inputs are then fed through these layers in a "forward" function. Pytorch then takes care of defining the relevant back-propagation algorithm.

Due to the extra complexities of the policy network it is not as simple to define this. much like for the value and soft Q networks we have a constructor that defines the structure of the network. The main difference her is that the final layer of the policy network is split into two. We have a linear function for the mean and a linear function for the standard deviation, each with an output size of $m := \dim \mathcal{A}$, which is why we have an output size $2m$ as described in section 3.3.2 and seen in figure 2. We also define bounds for the standard deviation here, as while we do want to explore the action space we don't want to deviate too far from the mean.

The policy network also needs two additional functions. The evaluate function takes a state and returns an action and the log of the probability of this action. This is done by feeding the state forward through the network to get the mean and standard deviation. We then sample from a Gaussian distribution with this mean and standard deviation, finally passing this through tanh to get our action. We then get the log of the probability of this action plus some noise. This function is used when calculating loss functions. The get_action function does the same thing as evaluate, however we don't calculate the probability as this function is used when we want to get an action to apply to the robot and the probability isn't needed here.

4.2 Agents

In order to keep the training as general as possible so that it can be used for any learning problem, I create an agent. This is an object that creates and contains all of the neural networks and parameters for learning as well as handling the update steps required in the algorithms. As SAC1 and SAC2 have different required parameters and networks, a separate agent class is needed for each.

The SAC1 agent requires a value network, target value network, two Q networks and a policy network as described in sections 3.3.2 and 4.1. We also initialise our target value function by copying the weights from the regular value function. When calculating the loss our Q and value networks we use the MSE loss function. While

this is easy to implement in python, there is a pre-built version in the package torch.nn, which will be faster due to its implementation. The last things that we need to include are the optimisers for each parameter. As mentioned previously, Haarnoja *et al.* [4] suggest using the Adam optimiser [8] for all parameters. Hence we create an instance of the Adam optimiser for each parameter except our target value function.

In the update function we take a random sample from the replay memory and convert these to tensors. We then calculate the value of each Q function and the value function at each state, action pair from the sample. We also sample the next actions and their probabilities from the policy network, given the next states. We use these to calculate the loss functions as described in section 3.2. We can then back-propagate the errors and step our optimisers to update the networks. We finally update the target value function as described in algorithm 1.

Similarly our SAC2 agent requires two Q networks and a policy network, however instead of a target value network we need two target Q networks and we have a variable α rather than a value network. As in the SAC1 agent we create instances of MSE loss for out Q networks and Adam optimisers for each parameter. The final thing we need for the SAC2 agent is our target entropy. We set this as $-\dim \mathcal{A}$ as suggested by Haarnoja *et al.* [4].

Here the update function is near identical to that in the SAC1 agent, however using the updated loss functions, equations 8, 9 and 10. Much like with the target value function for SAC1, we update the target Q functions as in algorithm 2.

We also add a function to both agents that saves the state of the policy network. This allows us to actually use the trained policy outside of training it.

I create the agents to be as general as possible so that they can be applied to any reinforcement learning problem that has been encapsulated in an OpenAI Gym [6] environment.

4.3 Training Loop

Now that we have an agent we can begin training our model. To do this we must create a training loop, in which our model will train. This function needs to be given as input an agent, an environment in which this agent will train, the maximum total number of steps to be taken, the maximum number of steps per episode and the batch size. The batch size is how large a sample we take at each update step in the algorithm.

At each step in the simulation we get an action and apply this to the environment to get the next state, the reward for the state action pair and whether the simulation has reached an end state. For the first $2 * batch\_size$ steps we randomly sample the action, allowing the replay memory to start to be filled and to allow the policy network to have some data fed through before it is sampled. After this point, however, we get the action from the policy network. The 5-tuple (state, action, next state, reward, done) is then pushed to the replay memory. After each step, we then

update the agent if the size of the replay memory is larger than the batch size.

This is repeated up to max steps times or until an end state is reached to make an episode. At the end of each episode we save the total episode reward to a list. If the total number of steps taken is less than the maximum total number of steps, we restart the environment and start a new episode. Once done the maximum number of steps we return the list of all episode rewards.

5 Testing and Evaluation

To train the model I use Google Colab [10] with graphics acceleration enabled. This allows for the model to be trained faster than on my own hardware, and for multiple models to be trained simultaneously. I train all models with 500,000 maximum steps. I find that this gives the algorithm time to find an optimal solution and to start reinforcing it. I choose 500,000 also, due to the fact it takes approximately 2.5 hours to train which is a reasonable period of time and similar to the results found by Haarnoja *et al.* [7]. I use 1000 steps per episode as the reward function is based on 1000 steps, so this should give the best possible reward. The batch size used for training is 256 as suggested by Haarnoja *et al.* [4].

To test the results of training, I record the reward of each episode. This shows how the model has trained over the training period, however since the algorithms encourage exploration of the action space we expect dips and spikes. To counteract this, I also plot the 50 epoch rolling average of the reward, which should show the trend of the learning. Another important statistic is the standard deviation of the learning. We want, by the end of the training, the standard deviation to be as low as possible as this indicates that the performance of the model is likely to be close to the average. Hence, I plot the 50 walk rolling standard deviation as well.

5.1 SAC1

Training with the SAC1 algorithm yields poor results. Figure 3 shows the training reward of with shake weight and energy weight kept constant at 0 and drift weight varied. We can see that for all trained policies the average reward stays between 1 and 2 throughout the training period, likewise for the standard deviation. This suggests that the there are frequent episodes that achieve 0 reward. We can also see that the spikes where a better than average episode occurs, achieve rewards of around 6 maximum. The results for varying drift weight with a constant energy weight of 0.005 and for keeping drift weight constant at 0, energy weight at 0 or 0.005 and varying shake weight, were similar to those shown in Figure 3.

Upon viewing the gaits achieved by the final policy of each of these, I find they have all achieved a similar gait. The policies have learned to move the front two legs in unison and the back legs in unison. It first does a small hop with the front legs, followed by a small hop with the back legs. This is repeated, resulting in a slow shuffling movement, until eventually it falls.

This suggests that the SAC1 algorithm isn't suitable for high dimensional learning problems, such as quadrupedal locomotion, in a sample efficient manner. It may

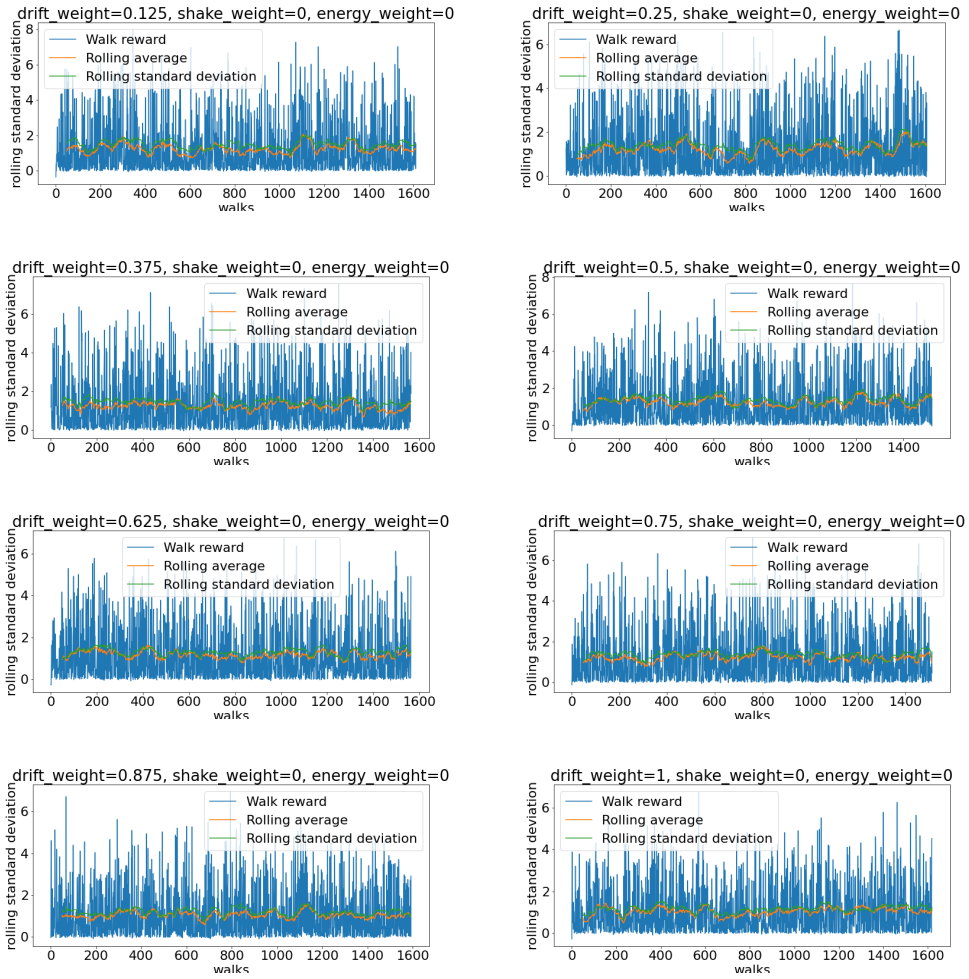


Figure 3: Varying drift with SAC1. Blue is the reward of each epoch, orange the 50 walk rolling average and green the 50 walk rolling standard deviation.

be the case that the algorithm needed more learning time to be able to develop a suitable gait. However, the fact that all of the experiments seemed to converge to a similar gait, suggests that this is not the case. The issue may also be elsewhere, for example tuning the learning rates used could give different results.

5.2 SAC2

The results of training on SAC2 were drastically different to training on SAC1.

5.2.1 Drift

In tuning drift with penalty for energy usage, I find that for lower and middle penalties we achieve a high expected reward by the end of training, as seen in Figure 4. We see that we have a fairly high reward for a drift weight of 0.25, however this drops for 0.375 and then starts to pick up again for 0.5 and 0.625. This seems to be because the lower weights do not penalise poor behaviour enough, so

we achieve a high reward for policies that don't manage to walk in a straight line. We can see that we achieve a similar reward for weights of 0.25 and 0.625, however for a weight of 0.25 we have that the standard deviation is higher in relation to the average reward than it is for 0.625. This would suggest that we obtain a more consistent gait for a weight of 0.625. For higher weights than this we have quite a low reward, though this could just be due to the fact that the penalty is large.

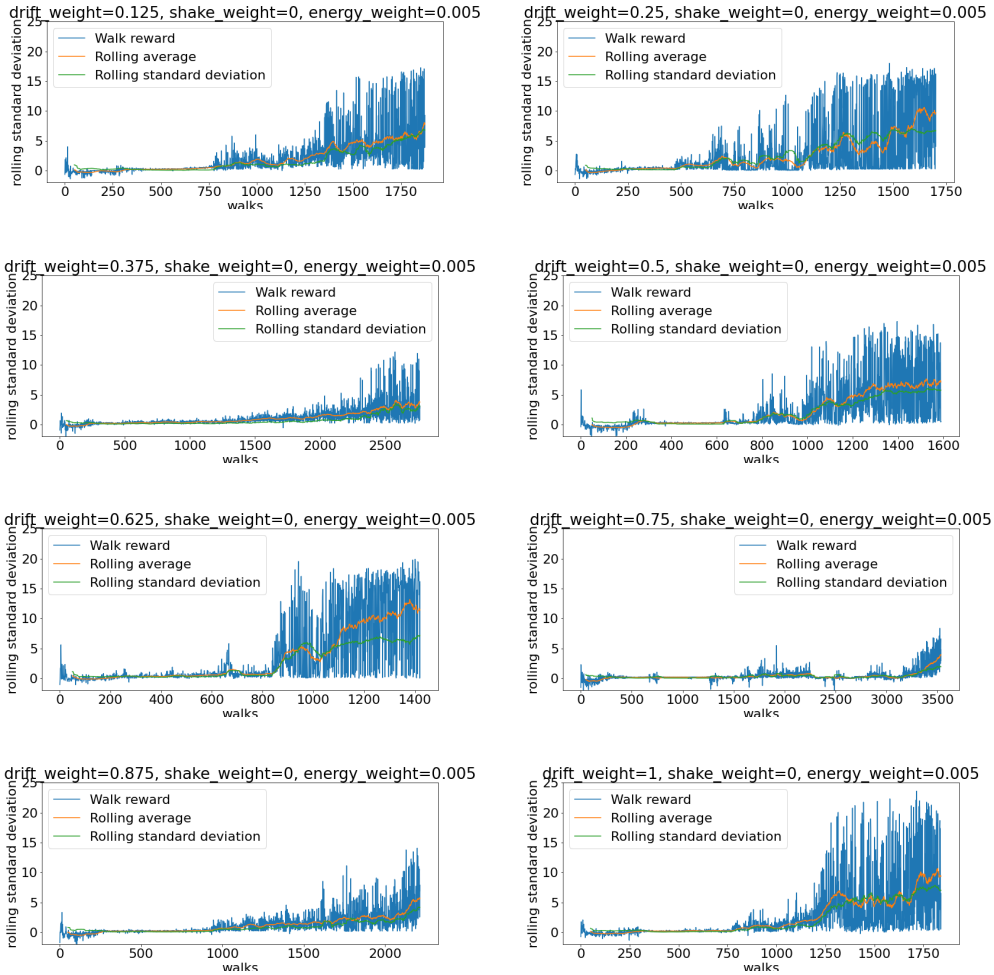


Figure 4: Varying drift with SAC2 and penalty for energy consumption. Blue is the reward of each epoch, orange the 50 walk rolling average and green the 50 walk rolling standard deviation.

To better compare these policies I run them in environments with a no penalty reward and full penalty reward, i.e. $drift\_weight = 1$, $shake\_weight = 1$, $energy\_weight = 0.005$. Upon doing this I find that for no penalty environments 0.625 drift weight performs the best followed by 0.25. With full penalty reward I find that again a weight of 0.625 performs best however, this time followed by 1. It is to be expected that 1 would perform well here as the full penalty environment is similar to what it trained on. This all suggests that with an energy penalty a drift weight of 0.625 is best.

We have a similar trend with training reward for varying the drift weight but with no energy penalty. However, where before we had that some policies had little to no reward at the end of training, here they all achieved an average reward of at least 10, as seen in Figure 5.

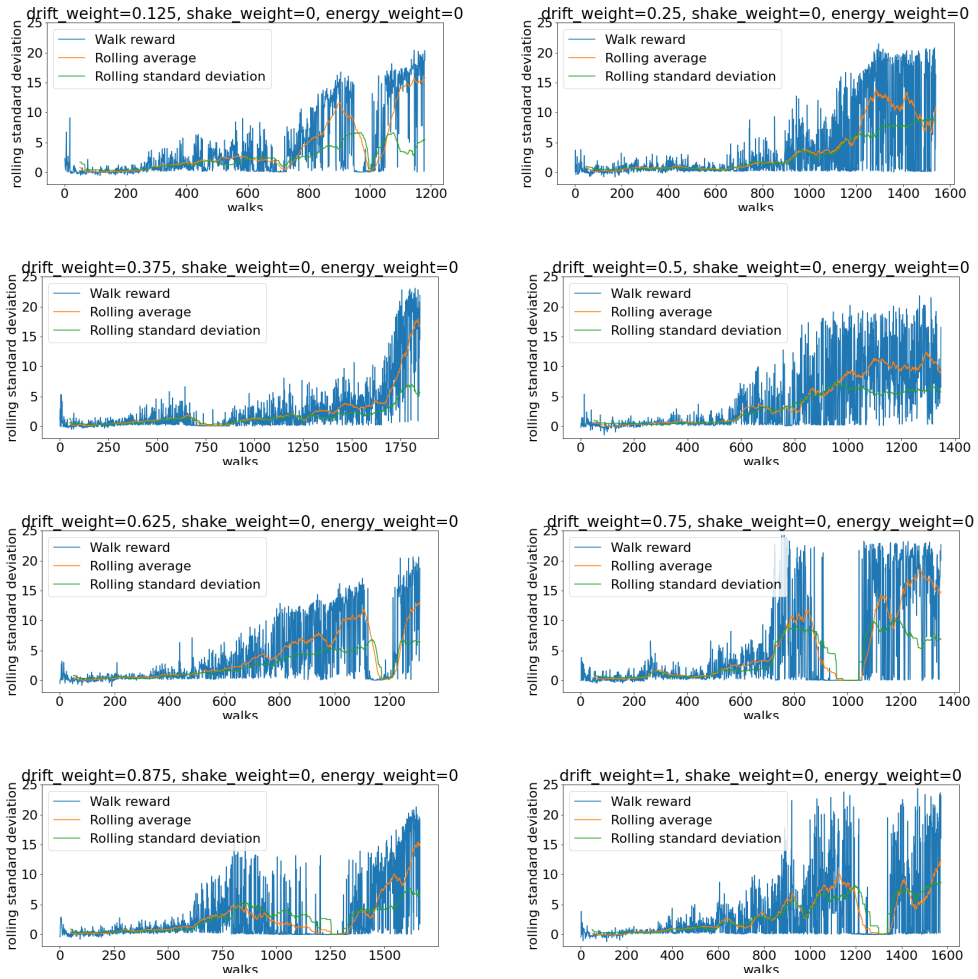


Figure 5: Varying drift with SAC2 and no energy penalty. Blue is the reward of each epoch, orange the 50 walk rolling average and green the 50 walk rolling standard deviation.

As before I test these on full penalty and no penalty reward environments to find their average reward return in these. I find that with no penalties 0.75 weight performs the best, followed by 0.375 and for full penalty, 0.875 performs the best followed by 0.75. This suggests that the 0.75 is the best drift weight for 0 energy weight. Upon reviewing the walking gait of this I find that it performs very well. At the beginning of the walk the robot turns slightly and then walks in a straight line for the rest of the period as seen in Figure 6.

5.2.2 Shake

The results for varying the shake weight are considerably worse than those for drift. We see inn Figures 7 and 8 that either very little reward is attained or the rolling

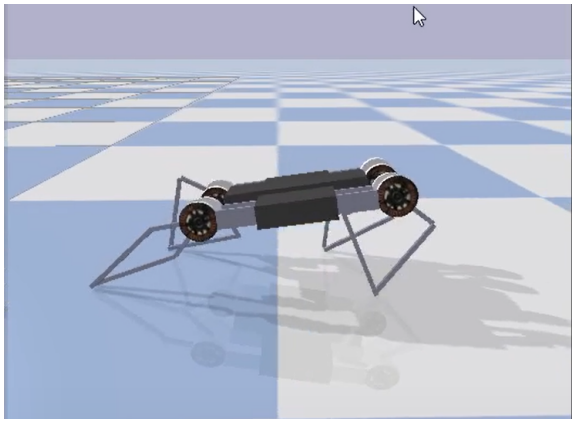


Figure 6: Walk with drift weight = 0.75, shake weight = 0 and energy weight = 0

standard deviation of the reward is very close to the rolling average, implying that a considerable amount of epochs resulted in very low reward.

For the shake policies trained with energy penalty I find that in a no penalty environment, shake weight of 0.5 performs best with 0.375 next. This was also the case for a full reward environment. Hence with energy penalties a medium shake penalty performs best.

Similarly, I find that with a no penalty reward for policies trained with no energy penalty, a shake weight of 0.625 performs the best followed by 0.375. For a full penalty reward 0.625 was again the best with a reward of 8.9, however the rest of the policies performed poorly receiving less than 5 average reward.

5.2.3 Energy

I find that on some policies trained with an energy penalty, we get a repeated pattern. With an energy penalty of 0.005 we see that a to minimise energy usage and hence maximise reward, the algorithm learns to not use one of its back legs. This leg gets dragged behind as the other legs walk forward. However as a consequence of the leg being dragged behind, the robot begins to turn. For example if the robot learned to not use the back left leg, it would start to turn left. Doing this reduces the forward attained at each step. To compensate for this the robot then starts to lean forwards on the opposite side to the leg being dragged, straightening out its path so it now walks diagonally. In the same example as before this would lean on its front right side. This behaviour can be seen in Figure 9.

A trend I find with the policies trained with no energy penalty is that they tend to have a more chaotic gait. This can be seen in how the legs move, their speeds and where they are being placed. the legs move quite quickly and the placement of the legs doesn't seem to follow a pattern as we would expect. This would likely be a suboptimal or even infeasible solution if applied to a real life robot. This is because the motor speeds that the policy learns may not be possible on the real robot, also the stress this causes may cause damage to the the robot that harms performance in the future.

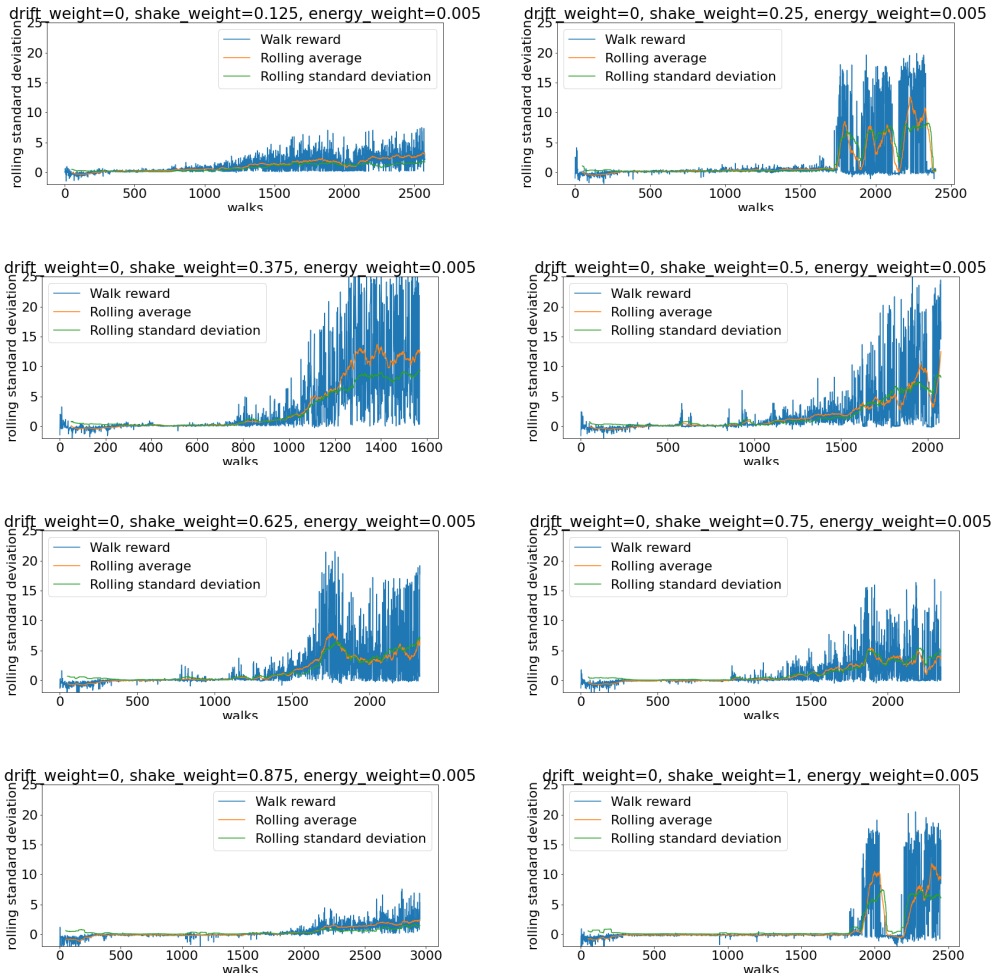


Figure 7: Varying shake with SAC2 and penalty for energy consumption. Blue is the reward of each epoch, orange the 50 walk rolling average and green the 50 walk rolling standard deviation.

On multiple policies that were trained with reward in the energy penalty I find behaviour similar to that found in nature. When walking the robot pushes forward with its legs then lifts them all almost simultaneously to place them further forward, this process is then repeated. This is almost like a horse' gallop.

5.3 General results

In a number of the training periods, particularly in Figure 5, we can see that the performance of the model improves and then suddenly drops to 0 and then picks up again. This strange behaviour seems to suggest that the SAC algorithm is fairly unstable and perhaps longer training periods would not improve the results but may even harm them. However this could be a problem that reducing the size of the replay memory may solve. If the memory size is too large it will still include early state transitions with low reward. When the replay memory is then sampled in the update step, there is a good chance that these early state transitions are sampled

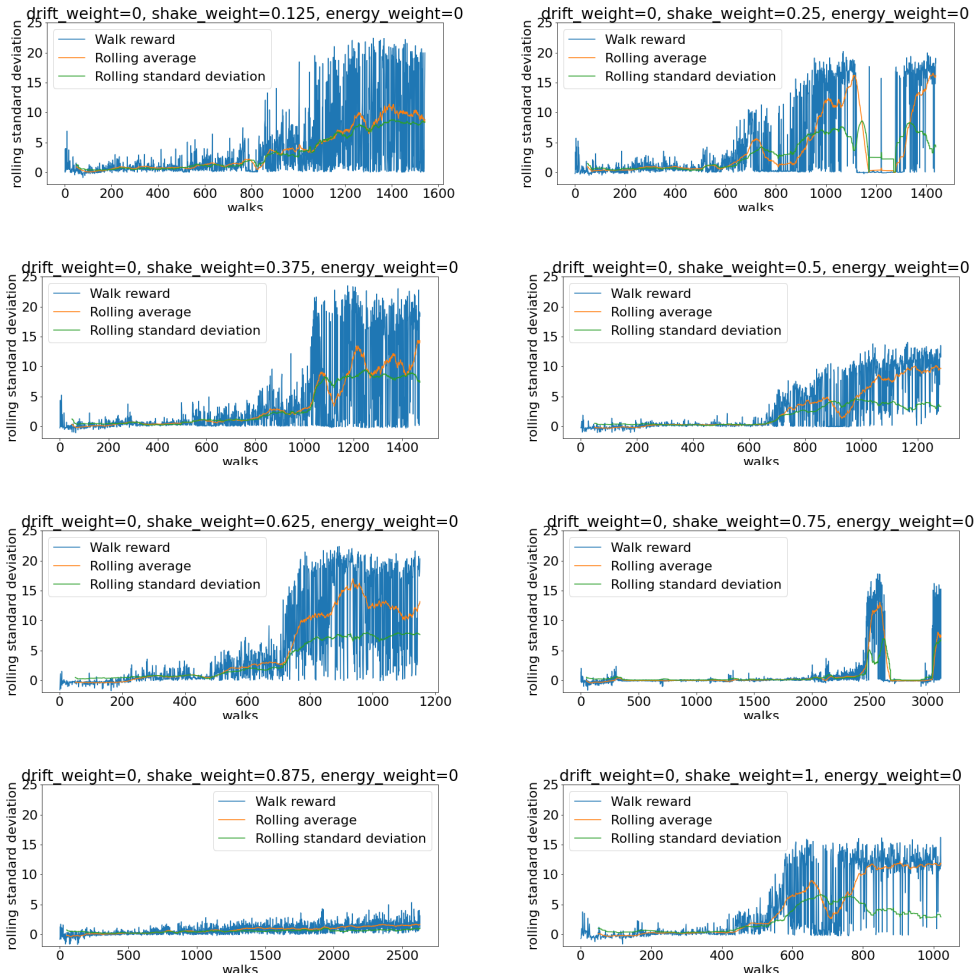


Figure 8: Varying shake with SAC2 and no energy penalty. Blue is the reward of each epoch, orange the 50 walk rolling average and green the 50 walk rolling standard deviation.

and used to update the neural networks.

6 Conclusion

I find that the SAC1 algorithm is unsuitable for learning the high dimensional problem of teaching a quadrupedal robot to walk in a short space of time. My results for the SAC2 algorithm show that the best results are achieved when the reward function has an energy penalty applied to it. In such cases we often get gaits that resemble those found in nature, such as a horse' gallop. I also find that the SAC algorithm sometimes has periods where the performance of the policy dips for large periods of time in training quite suddenly, suggesting that the learning stage may be unstable.

I investigate which weights in the reward function of the environment result in the best learned policy. However I only investigated each weight separately not the

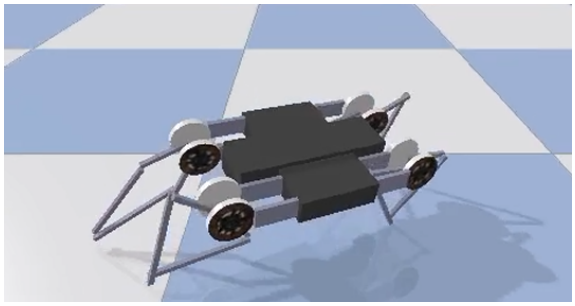


Figure 9: Back left leg is dragged behind causing a lean towards the front right.

wider space where they interact with each other. This wider exploration of the space of combinations of drift and shake weights could reveal better results from a SAC approach.

Since my agents were as created to be general so that any OpenAI Gym environment could be learned through my implementation, not just minitaur walking, I could have used the Gym Benchmarking problems. This would allow me to evaluate my method against other reinforcement learning methods.

Future work on this project could include further exploring how changing the parameters in the reward function alter the results of learning. A possible downside of the reward function used in the environment that was used to train the policies is that it also punishes course correction. If the robot does go off course at any point, e.g. by turning or drifting, since the reward function penalises for any movement of this kind, were it to then try to turn in the other direction to get back to the would be discouraged. As such a better reward function that takes into account the where the robot is moving to rather than just how it is moving and encourages course correction, may result in better results. A similar approach could also be used to create reward functions that encourage walking in certain directions and from this an autonomous controller could start to be developed.

References

[1] J. Tan, T. Zhang, E. Coumans, A. Iscen, Y. Bai, D. Hafner, S. Bohez, and V. Vanhoucke, "Sim-to-Real: Learning Agile Locomotion For Quadruped Robots," *Robotics: Science and Systems XIV*, apr 2018. [Online]. Available: http://www.roboticsproceedings.org/rss14/p10.pdfhttp://arxiv.org/abs/1804.10332

[2] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. Van Den Driessche, T. Graepel, and D. Hassabis, "Mastering the game of Go without human knowledge," *Nature*, vol. 550, no. 7676, pp. 354–359, 2017. [Online]. Available: http://dx.doi.org/10.1038/nature24270

[3] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor,"

35th International Conference on Machine Learning, ICML 2018, vol. 5, pp. 2976–2989, jan 2018. [Online]. Available: http://arxiv.org/abs/1801.01290

[4] T. Haarnoja, A. Zhou, K. Hartikainen, G. Tucker, S. Ha, J. Tan, V. Kumar, H. Zhu, A. Gupta, P. Abbeel, and S. Levine, "Soft Actor-Critic Algorithms and Applications," 2018.

[5] E. Coumans and Y. Bai, "2019. PyBullet, a Python module for physics simulation for games, robotics and machine learning," 2019. [Online]. Available: https://pybullet.org/wordpress/

[6] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "OpenAI Gym," pp. 1–4, jun 2016. [Online]. Available: http://arxiv.org/abs/1606.01540

[7] T. Haarnoja, S. Ha, A. Zhou, J. Tan, G. Tucker, and S. Levine, "Learning to Walk via Deep Reinforcement Learning," *Robotics: Science and Systems XV*, dec 2018. [Online]. Available: http://www.roboticsproceedings.org/rss15/p11.pdfhttp://arxiv.org/abs/1812.11103

[8] B. Ziebart, "Modeling Purposeful Adaptive Behavior with the Principle of Maximum Causal Entropy," Ph.D. dissertation, 2010. [Online]. Available: papers2://publication/uuid/AEB1E579-7EC8-4DAD-8B3C-FFE66E8B314F

[9] D. P. Kingma and J. L. Ba, "Adam: A method for stochastic optimization," in *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*, 2015.

[10] Google, "Google Collaborative." [Online]. Available: https://colab.research.google.com/