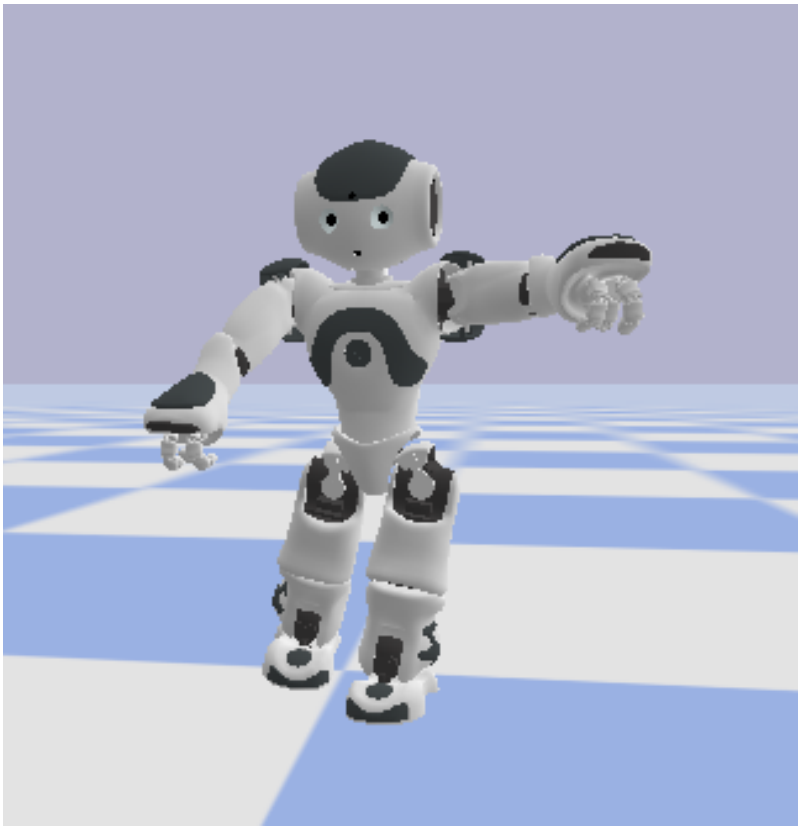
Teaching a Nao robot to balance on one leg using reinforcement learning



Junis Nagel

Layout: typeset by the author using LaTeX.

Teaching a Nao robot to balance on one leg using reinforcement learning

Exploring the capabilities of the Soft Actor Critic algorithm

Junis Nagel
13448714

Bachelor thesis
Credits: 18 EC

Bachelor *Kunstmatige Intelligentie*

University of Amsterdam
Faculty of Science
Science Park 900
1098 XH Amsterdam

Supervisor
Dr. A. Visser

Informatics Institute
Faculty of Science
University of Amsterdam
Science Park 900
1098 XH Amsterdam

June 28, 2024

Abstract

This thesis investigates the application of the Soft Actor-Critic (SAC) algorithm to enable a Nao robot to balance on one leg, a crucial step towards executing dynamic kicking motions. The current approach of the Dutch Nao Team involves a static kick using a fixed sequence of joint angles without active balance control. In contrast, this research aims to develop a reinforcement learning (RL) based method leveraging the SAC algorithm, known for its effectiveness in continuous action spaces and robustness to hyperparameters.

Contents

Chapter 1

Introduction

1.1 Problem statement

The Dutch Nao Team is a team of UvA students who are competing in the RoboCup, a competition where teams of humanoid robots (Nao robots) autonomously play a soccer match against other teams. Numerous interconnected components are needed to accomplish this task, such as computer vision, a movement engine and a gameplay strategy. This thesis aims to lay a foundation for perform a dynamic kicking motion by focusing on the first crucial step of performing a kick, balancing on one leg. A purely reinforcement learning (RL) based approach will be attempted in contrast with existing methods that rely on, for example, bezier curves whose parameters are tuned for the different phases of the kicking motion. Currently, the Dutch Nao Team is using a static kick. This means that the kicking motion is always performed using a fixed sequence of manually selected joint angles. Since the sequence is fixed, there is no active balance control.

The aim of this paper is to explore the capabilities of the soft actor critic (SAC) algorithm for this task. SAC is an algorithm that combines elements of actor-critic methods as well as maximum entropy reinforcement learning. It has shown promise in domains with continuous action spaces such as robotics (Haarnoja, Zhou, Abbeel, & Levine, 2018) and has the advantage of being able to handle both deterministic and stochastic policies.

To achieve the goal of implementing a stable one legged stance, several key steps will be taken. Firstly, a simulation environment will be set up to allow for rapid parallel training and testing of different hyperparameters and reward functions. This environment will model the dynamics of the Nao robots and their interactions with the soccer ball and other objects on the field. Secondly, the SAC algorithm will be implemented and trained in the simulated environment. This training process will involve iteratively refining the reward function to improve performance

and adaptability and adjusting the hyperparameters of the SAC model to aid in rapid training.

Bipedal robots like the Nao are inherently rather unstable due to their relatively high center of mass, and dynamic movements such as kicking can therefore introduce additional complexities. To address this challenge, noise such as random pushes will be introduced in the training phase to force the learning algorithm to build a more robust model of the world and learn to adapt to changes in its environment.

1.2 Related work

As briefly mentioned in the introduction the challenge of creating a dynamic kick for the Nao robot has been tackled before with varying degrees of dependency on RL. Some papers utilize manually created controllers (Draskovic, 2022), while others incorporate RL to learn model parameters such as kicking speed and body orientation (Rezaeipanah, Amiri, & Jafari, 2021). There are also parameterized models where the parameters, such as control points for splines (Engelke, 2021) are learned through RL and there are purely reinforcement learning based approaches where the joint velocities or target positions are directly controlled by a RL model (Abreu, Silva, Teixeira, Reis, & Lau, 2021). The latter is the approach taken in this paper.

For the task of balance control specifically imitation learning has been applied before (Bong, Jung, Kim, & Park, 2022), but was considered to be a too large scope for this paper. Balance control can be done through hip and ankle joint adjustments or with the help of the robots' arms. The arms can be particularly helpful for reacting to external forces (Nakada, Allen, Morishima, & Terzopoulos, 2010).

Erez and Smart propose shaping as a means to improve reinforcement learning algorithms. They suggest multiple six ways in which this can be done, three of which will be applied in this paper. Namely modifying the reward function, modifying internal parameters and modifying the action space.

Chapter 2

Theoretical Background

2.1 Soft Actor-Critic Algorithm in Stable Baselines 3

2.1.1 Introduction

Reinforcement Learning (RL) has seen significant advancements with algorithms like Deep Q-Networks (DQN), Proximal Policy Optimization (PPO), and Soft Actor-Critic (SAC). Among these, SAC stands out for its ability to handle continuous action spaces efficiently while promoting exploration through entropy maximization (Haarnoja et al., 2018) instead of using an epsilon-greedy strategy, as seen in the algorithms mentioned above. This section provides a review of the SAC algorithm as implemented in the Stable Baselines 3 (SB3) python library.

2.1.2 Background and Motivation

Traditional RL algorithms can struggle with the trade-off between exploration and exploitation, particularly in high-dimensional continuous action spaces (Dulac-Arnold et al., 2021). SAC addresses this challenge by incorporating an entropy term, which encourages the agent to explore a wider range of actions by maximizing the entropy of the policy. This approach not only improves exploration but also enhances the stability and robustness to hyperparameters and environmental changes. Another advantage of SAC is it's sample efficiency, allowing for rapid prototyping .

2.1.3 Algorithm Overview

The SAC algorithm optimizes a stochastic policy $\pi_\theta(a|s)$ to maximize the expected reward while also maximizing the entropy of the policy. The objective function for SAC is:

$$J(\pi) = \mathbb{E}_{(s_t,a_t)\sim\rho_\pi} \left[Q_\pi(s_t, a_t) - \alpha \log \pi(a_t|s_t) \right], \tag{2.1}$$

where $Q_\pi(s_t, a_t)$ is the Q-value under policy π, and α is a temperature parameter that balances reward and entropy.

2.1.4 Algorithm Evolution

The initial versions of SAC included a value function $V_\psi(s)$ alongside the Q-functions $Q_{\theta_1}(s, a)$ and $Q_{\theta_2}(s, a)$. The value function reduced variance in Q-value updates but added computational complexity. The modern version simplifies this by directly using the Q-functions, which improves efficiency.

Older Version with Value Function

In the older SAC version, the value function $V_\psi(s)$ was updated using:

$$J_V(\psi) = \mathbb{E}_{s_t\sim\mathcal{D}} \left[\frac{1}{2} \left(V_\psi(s_t) - \mathbb{E}_{a_t\sim\pi_\theta} \left[Q_\theta(s_t, a_t) - \alpha \log \pi_\theta(a_t|s_t) \right] \right)^2 \right], \tag{2.2}$$

The policy was updated by minimizing the Kullback-Leibler divergence between the policy and the exponentiated Q-function. The Kullback-Leibner divergence is a measure to quantify the statisctical difference between two probability distributions.

Modern Version without Value Function

The modern SAC implementation in SB3 uses two Q-functions without an explicit value function. The Q-value update is performed using the minimum of the two Q-values:

$$J_Q(\theta_i) = \mathbb{E}_{(s_t,a_t,r_t,s_{t+1})\sim\mathcal{D}} \left[\frac{1}{2} \left(Q_{\theta_i}(s_t, a_t) - y_t \right)^2 \right], \tag{2.3}$$

where

$$y_t = r_t + \gamma \mathbb{E}_{a_{t+1}\sim\pi_\theta} \left[\min_{i=1,2} Q_{\theta_i}(s_{t+1}, a_{t+1}) - \alpha \log \pi_\theta(a_{t+1}|s_{t+1}) \right]. \tag{2.4}$$

The policy update maximizes the expected Q-value minus the entropy term:

$$J_\pi(\theta) = \mathbb{E}_{s_t \sim \mathcal{D}, a_t \sim \pi_\theta} \left[\alpha \log \pi_\theta(a_t|s_t) - \min_{i=1,2} Q_{\theta_i}(s_t, a_t) \right]. \tag{2.5}$$

2.1.5 Implementation Details in Stable Baselines 3

SB3 has implemented this newer version of SAC with some specific details to enhance performance and usability:

Policy Network

The policy network in SAC outputs the parameters of a probability distribution from which actions are sampled. The mean and standard deviation are learned separately, with the standard deviation parameterized as the logarithm of the standard deviation to ensure positivity.

Squashing Function

A squashing function, usually a tanh activation, is applied to the output of the policy network to ensure actions remain within valid bounds. The entropy calculation is adjusted to account for the change in distribution due to the squashing function:

$$\log \pi(a|s) = \log \pi_{\text{raw}}(a_{\text{raw}}|s) - \sum_i \log \left(1 - \tanh^2(a_{\text{raw},i})\right), \tag{2.6}$$

where π_{raw} represents the probability distribution before applying the tanh function.

Target Networks

SAC uses target networks for the Q-functions to stabilize training. These target networks are updated using a soft update mechanism:

$$\theta_{\text{target}} \leftarrow \tau\theta + (1 - \tau)\theta_{\text{target}}, \tag{2.7}$$

where τ is a small value to ensure smooth updates.

2.1.6 Algorithm Outline

Algorithm 1: Soft Actor-Critic Algorithm

Initialize Q-networks Q_{θ_1}, Q_{θ_2}, policy π_θ, and target networks Q'_{θ_1}, Q'_{θ_2} with $\theta' \leftarrow \theta$.
Initialize replay buffer $\mathcal{D}$.
for every iteration **do**
  for every step **do**
    Sample action $a_t \sim \pi_\theta(a_t|s_t)$.
    Execute action a_t in environment, observe reward r_t and next state s_{t+1}.
    Store transition (s_t, a_t, r_t, s_{t+1}) in $\mathcal{D}$.
  end for
  for each gradient step **do**
    Sample batch of transitions (s_t, a_t, r_t, s_{t+1}) from $\mathcal{D}$.
    Compute target Q-value:
    $y_t = r_t + \gamma \mathbb{E}_{a_{t+1} \sim \pi_\theta}[\min_{i=1,2} Q'_{\theta_i}(s_{t+1}, a_{t+1}) - \alpha \log \pi_\theta(a_{t+1}|s_{t+1})]$.
    Update Q-networks by minimizing $J_Q(\theta_i)$.
    Update policy by maximizing $J_\pi(\theta)$.
    Soft update target networks: $\theta' \leftarrow \tau\theta + (1-\tau)\theta'$.
  end for
end for

Chapter 3

Method

Balancing is challenging task. Multiple approaches were taken to address the challenge. To validate the choice of the SAC implementation by SB3 in combination with qiBullet the cartpole problem was tackled. This is a much simpler task than balancing on one leg, but can provide some confidence in an entropy maximizing based approach to balancing. First, the common parts of the different tasks are outlined, followed by specific information for each task.

3.1 Simulator

The use of a simulator was chosen over the use of a physical robot to allow for quicker and parallel testing while also providing access to more variables such as the absolute foot height. In order to be able to accurately use this variable in the reward function outside of a simulator the use of, for example, cameras and object detection would be needed. This complicates the technical setup without providing benefits to the research. The chosen simulator is qiBullet, an open-source simulator by United Robotics. Its goal is to provide a physically accurate simulation tool via a Python based API (Aldebran Robotics, n.d.). The original Bullet simulator uses a C++-based API, which makes it incompatible with most reinforcement learning frameworks and libraries such as Tensorflow, SB3 and OpenAI Gym. Higher-level languages like Python offer greater flexibility and ease in testing new approaches quickly. qiBullet is built upon PyBullet (Busy & Caniot, n.d.), a python module that interfaces with the Bullet Physics SDK, making it compatible with the aforementioned frameworks and libraries.
The simulator has a step frequency of 240 per second. The URDF file for the cartpole task and ground plane are available in pyBullet and the URDF file for the Nao robot is provided in qiBullet. While qiBullet provides methods to read the simulated IMU and FSR sensor data for the Nao robot, these were not used

and a custom function was implemented instead. More on this can be found in discussion section 6.1.2 and the corresponding code used instead can be found in appendix section 7.1.

3.2 Environment

OpenAIs Gymnasium provides a standardized API for reinforcement learning libraries and frameworks, such as Stable Baselines3, to interact with. The base class can be extended to create a custom environment which can be passed to algorithm implementations of SB3. The main methods and attributes that the environment need to have are outlined below.

3.2.1 Methods and attributes

Step()

This method takes as argument the action to be taken and is called to apply this action to the environment and perform a single simulation step. It returns a new observation vector, the reward for the new state, a boolean indicating if the current episode is finished (in the case of this paper this would be when a fall is detected), a boolean indicating if the current episode is truncated (the maximum episode length is reached), and a dictionary for logging purposes.

Reset()

The reset method resets the environment and its variables to its initial state. PyBullet has a method to reset the simulator, but that reloads the robot from the URDF file. A function was implemented instead to reset the robots position, orientation and joint angles and velocities. It also returns the observation vector and a dictionary for logging purposes.

Action space

The action space defines the set of actions the agent can explore. Any action vector calculated by the SAC algorithm should fall within this action space.

The action space used by SB3 was normalized to lie between -1 and 1 for every dimension. Remapping to the actual action space of the joints was done using formula 3.1 for every dimension. This formula is suitable for non-symmetrical action spaces and ensures a remapping of 0 to the mean:

$$0.5 * (\text{action} + 1) * (\text{upper limit} - \text{lower limit}) + \text{lower limit} \tag{3.1}$$

The upper and lower limit depend on the control mode (position or velocity). The action space has type spaces.Box from the gymnasium library.

Observation space

An observation vector represents the state of the environment, or in this case, the state of the robot. The observation space was also normalized by using the inverse of the previous formula:

$$2 * \frac{\text{observation} - \text{lower limit}}{\text{upper limit} - \text{lower limit}} - 1 \tag{3.2}$$

This maps the observation to the range -1 to 1, with the mean mapped to 0. Like the action space, this space also has type spaces.Box from the gymnasium library.

3.3 Reward function

The reward functions used in this thesis are comprised of multiple components R, each with its own multiplier α to allow for manual adjustment of their influence on the final reward for a given state. By setting multiplier α_i to 0 effectively removes the corresponding component R_i from the reward function. This allows for assessment of the influence of each component of the reward function on the performance of the model. Additionally, there are one or more conditions $f \in F$ that constitute a failure, such as falling over. Meeting one of these conditions result in penalty P_{fail} for that episode. The calculation of the final reward is calculated as follows:

$$R_{\text{final}} = \begin{cases} \sum_{i=1}^{n} \alpha_i R_i, & \text{if no fall is detected} \\ P_{fail}, & \text{if a condition } f \in F \text{ is met} \end{cases}$$

Explanations of the reward components for each task can be found in their respective sections.

3.4 Cartpole balancing

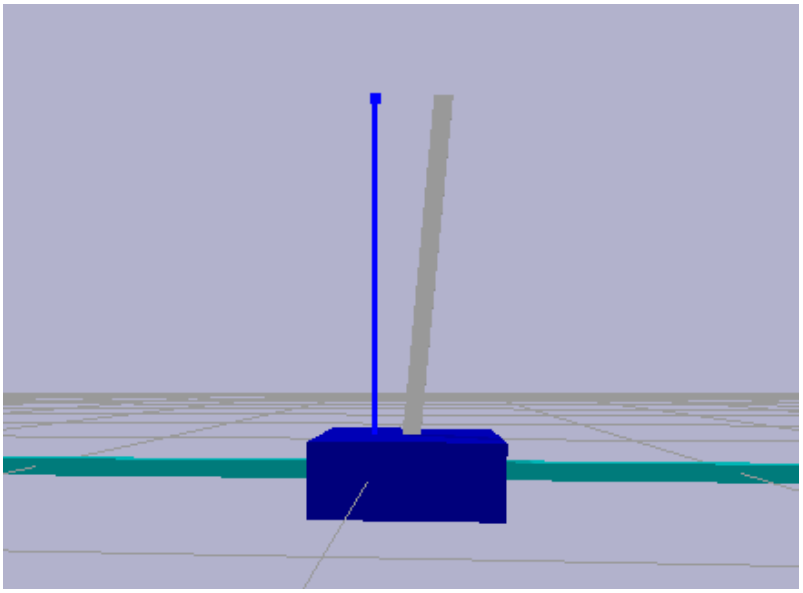


Figure 3.1: A visual representation of the cartpole task.

Figure 3.1 shows the cartpole balancing task. It is a relatively simple task where a 0.1 kg pole (white) is attached to a cart (blue) using a freely moving angular joint. The cart can slide along an axis (light blue) at a maximum velocity of 5 m/s to attempt to keep the pole from falling over. An additional goal is to keep the cart from moving too far away from the origin. The urdf file for this task is included in PyBullet and the simulations were ran for 500000 simulation steps, which the equivalent to approximately 35 minutes of real time practice.

3.4.1 Action and observation space

The action space for this task the velocity of the cart along the axis. The observation space is the velocity and the position of the cart along the axis and the angular position and velocity of the pole.

3.4.2 Reward function

COM (center of mass) distance

When the COM is located directly above the joint and no additional movements are made or external forces are applied, the robot is in a stable state. A negative

reward is therefore given relative to the distance of the projection of the COM with respect to the joint connecting the cart and the pole.

Failing conditions

The model fails when one of two conditions is met. Either the pole has an angular deviation of more than of twelve degrees relative to the upright position or the cart has a deviation of more than 0.4m from the origin. Both end the episode early.

3.5 Nao Robot balancing

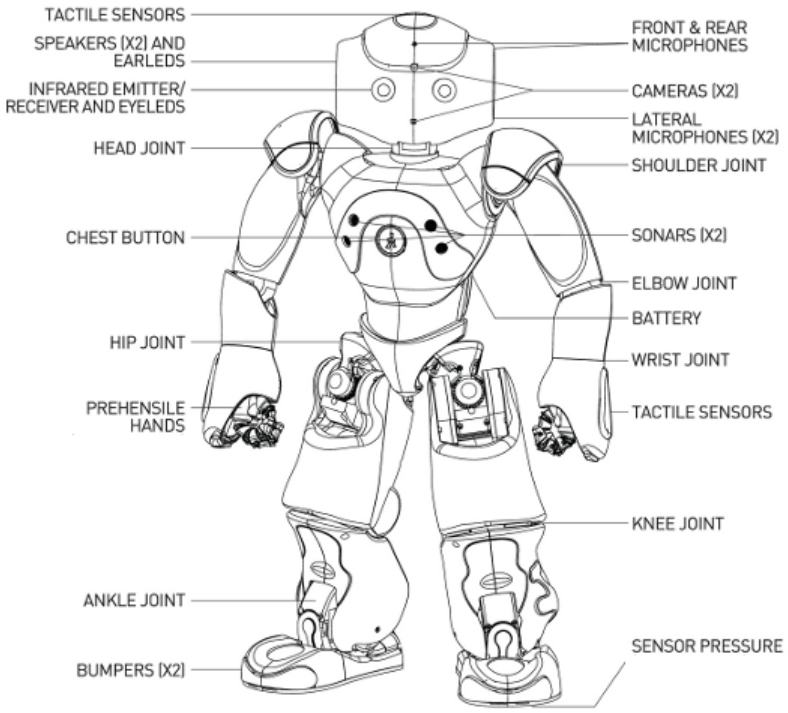


Figure 3.2: Specifications of the NAO H25 [1].

The robot used is the Nao H25 V5.0 robot by United Robotics (formerly SoftBank robotics and Aldebaran Robotics). This humanoid robot comes with a multitude of sensors built in, a three-dimensional gyroscope and accelerometer in its head and magnetic rotary encoders to enable reading of the joint angles [1].

[1] `http://doc.aldebaran.com/21/family/nao_h25/index_h25.html`

Additionally, there are four force sensitive resistors (FSRs) in four locations underneath each of its feet. Their locations are shown in figure 3.3 below.

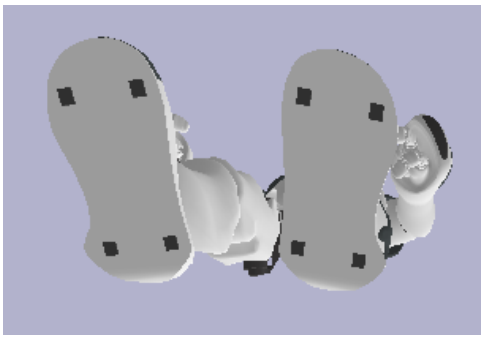


Figure 3.3: Black squares representing the FSR sensors and their locations.

3.5.1 Setup

The lateral as well as the spinning friction coefficient for the ground plane were set to 1, their default values could not be found in the pyBullet documentation. For every step in the training process five simulation steps are performed. Since the simulation frequency is 240 steps per second, there are $240/5 = 48$ motor velocity changes, state observations and rewards collections per second. This design choice was made to reduce erratic movement by the robot. Another choice for the same purpose was to limit the motor velocities to 10% of their respective maximum velocities.

3.5.2 Action and observation space

For the Nao robot, joint control is done through angles or velocities. To simplify the learning task and select the most crucial joints for the balancing task, the lifting foot ankle joints, head, elbow, wrist, hand and finger joints as disregarded in the action space. These joints likely won't contribute significantly to maintaining balance and complicate the learning process unnecessarily. The remaining joints are the action joints. Two different action spaces were explored in this paper, one in which the shoulder joints were part of the action joints and one in which they were not. This gives and action space dimensionality of 14 and 10 respectively.

The observation space contains the gyroscope and accelerometer readings and the joint angles and velocities of the action joints.

3.5.3 Reward function

The reward function for this task consists of five components, with an additional penalty P_{fail} when the robot falls or its supporting leg is too far of the ground. They are briefly explained below.

Foot height

The goal for the foot height y was set at 5cm, therefore a reward equal to the foot height up to 0.05 was given. This was later changed to

$$R_{\text{final}} = \begin{cases} y, & \text{y} <= 0.05 \\ 0.1 - y, & \text{y} > 0.05 \end{cases}$$

to discourage lifting the foot too high in the second round of training. More on why this change was made can be found in section 4.3.

Number of contact points reward

The number of contact points of the foot is determined by counting how many of the FSR values are non-zero. This reward is meant to encourage staying in contact with the ground. The resulting value is divided by four to normalize it and squared to provide a steeper gradient when more points are in contact with the ground plane.

COM above the support polygon reward

The support polygon is defined as the polygon described by the projection of the four positions of the FSRs (see figure 3.2) on the ground plane. When the COM is located directly above the support polygon and no additional movements are made or external forces are applied, the robot is in a stable state.

To simplify calculations and encourage a stance where the COM projection is closer to the center of the support polygon, the support polygon used is simplified and made to be smaller than the actual foot shape. This is a more stable position since it is easier for the projection of the COM to move outside of the support polygon upon movement or external factors when it is at the edge of the support polygon.
This is a binary reward, its value is either 0 or 1.

COM distance to the center of the support polygon penalty

The distance of the COM projection to the center of the support polygon is used to provide a smoother gradient in the reward function towards a stable one legged

stance.
The distance is used as a penalty. The distance is inherently a positive value, so it is multiplied by -1 to incentives minimizing the distance. It additionally promotes moving towards the center of the support polygon to stabilize the stance further. This is a more nuanced and gradual approach than the binary reward described above.

Torso orientation reward

This reward represents the cosine of the angle between the robot's z-axis and the world z-axis.

It is equal to 1 when the torso is completely upright and decreases towards 0 when the torso is horizontal. It is also cubed.

Failing penalty

An episode is considered a fail when one of two conditions is met. Either the torso orientation deviates too much from the upright position, or the supporting foot ankles is lifted more than 2 centimeters compared to its starting height. Falling is defined as having a torso orientation reward of less than 0.85, which corresponds to a torso z-axis orientation of more than 30 degrees compared to the world z-axis. The episodes is terminated early when either condition is detected and the other reward components are disregarded for this training step. The failing penalty is equal to -10.

3.6 Evaluation

The weights and biases (wandb) library provides a web based tool to visualize data collected by the SB3 logger and a custom wandb callback. The reward function components as described in section 3.4 are also logged for each time step to facilitate a clear overview of their scale and change in contribution over time. The main metric to evaluate model performance is the episode length, since an episode is terminated early when a failing condition is met. The average reward increasing is not a good indication for actual progress, since one of the reward function components might increase, such as keeping the torso upright for the Nao robot task, without also keeping the COM above the support polygon. This phenomena is called "reward hacking" and should be avoided. For this reason visual inspection of the learned policy is imperative to ensure that the learned policy performs as expected and no reward hacking occurs.

Chapter 4

Experiments and Results

4.1 Introduction

Several experiments were performed to evaluate the influence of the hyperparameters. The reward function parameters to be tuned are outlined in section 3.4. The model hyperparameters that were evaluated are the learning rate, batch size, gamma value and neural network breadth and width (denoted as [layer 0 size, layer 1 size...]). Initially, the model hyperparamaters were kept at their default value. Unless specified otherwise, reward function multipliers can be assumed to be 0 and model hyperparameters to be at their default value unless specified otherwise. The default hyperparamter values for SAC are outlined in table 4.1 below.

Learning rate	Gamma	Batch size	Model size
0.0003	0.99	256	[256, 256]

Table 4.1: Default values for tuned SAC hyperparameters

4.2 Cartpole task

4.2.1 Initial experiment

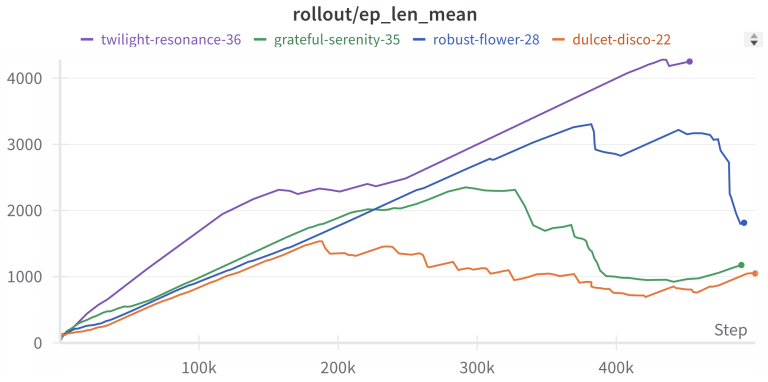


Figure 4.1: Mean episode length for different learning rate and gamma values.

Name	Gamma	Learning rate	COM multiplier	X deviation multiplier
twilight-resonance-36	0.99	0.001	10	0.1
grateful-serenity-35	0.98	0.001	10	0.1
robust-flower-28	0.99	0.01	10	0.1
dulcet-disco-22	0.98	0.01	10	0.1

Table 4.2: Hyperparameters corresponding to figure 4.1.

Both runs with a gamma value of 0.99 performed better than the other two. Twilight resonance with its lower learning rate managed to keep improving for longer. To verify that a lower learning rate is beneficial in this task another experiment was performed.

4.2.2 Final experiment

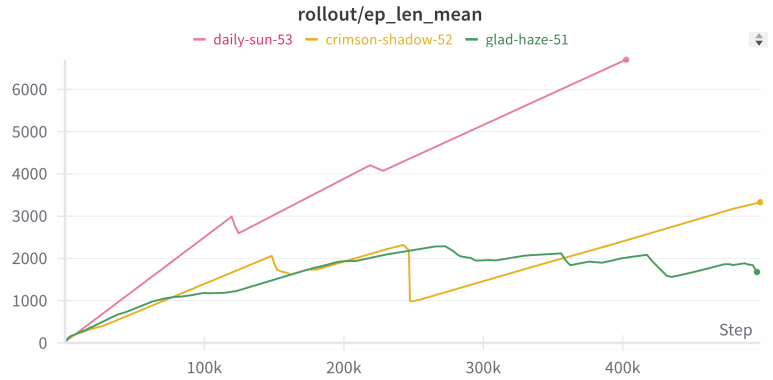


Figure 4.2: Mean episode length for different batch sizes and learning rates

Name	Learning rate	COM multiplier	X deviation multiplier
dail-sun-53	0.001	10	1
crimson-shadow-52	0.001	10	0.1
glad-haze-51	0.01	10	0.1

Table 4.3: Hyperparameters corresponding to figure 4.2.

The gamma value was set to 0.99 for every model. Rollouts are only calculated at the end of episodes. While daily-sun did complete all 500000 time steps, the final episode length was not logged as the model did not fail anymore before the end of the 500000 time steps. The last episode length was logged at time step 402384, which means the last episode lasted 97616 time steps before being truncated. This is equal to more than six and a half minutes.

While these results confirm that the lower learning rate enhances the models ability to keep learning over time. The biggest improvement however was due to the increase in the penalty for the deviation of the cart along the x axis.

This model was further tested by applying forces ranging from -25N to 25N for a single time step to the center of the pole every second in the horizontal direction, a task on which the model was not specifically trained. 25N equates to . It was able to stay balanced for over 10 minutes without falling before the episode was manually stopped.

4.3 Nao balancing task

The graphs in this sextion will have smoothing applied to their graph. The smoothing used is the running average with a box size of 100 samples [1]. Runs are assigned names random names by wandb, followed by the run number.

4.3.1 Initial experiments

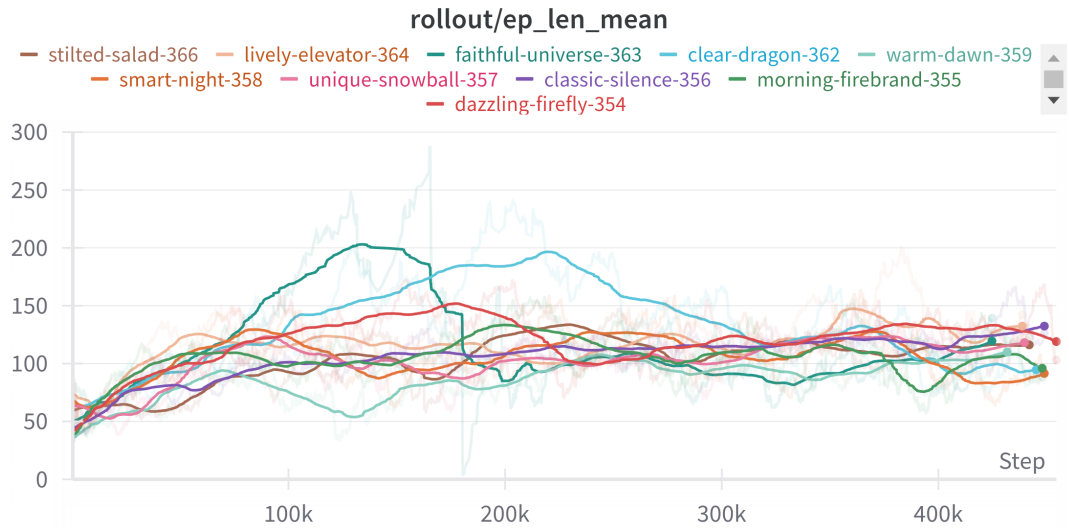


Figure 4.3: Smoothed episode length for initial experiments.

Name	Foot height	COM distance to support	Torso upright
stilted-salad-366	5	10	1
lively elevator-364	15	20	0.5
faithful-universe-363	15	20	0.5
clear-dragon-362	15	20	0.5
smart-night-358	15	20	0.5
unique-snowball-357	15	20	2
classic-silence-356	15	10	1
morning-firebrand-355	20	20	1
dazzling-firefly-354	15	20	1

Table 4.4: Model and reward function parameters for 4.3

[1] `https://docs.wandb.ai/guides/app/features/panels/line-plot/smoothing`

This experiment was supposed to run for 1.000.000 steps, but crashed and finished prematurely. Due to time constrains the experiment was not reran. The initial test shows invariance to reward function parameters, with the exception of the foot height multiplier. Faithful universe as well as clear dragon stand out for their initial jump in episode length around 100-200 thousand steps. The reward function parameters for these models are the same, and will be used for tuning of the model parameters in the hope of continuing their upwards trend without coming back down again.

4.3.2 Learning rate and gamma value with shoulder control added

Next the action space was expanded to include shoulder control and the learning rate and gamma parameter were assessed.

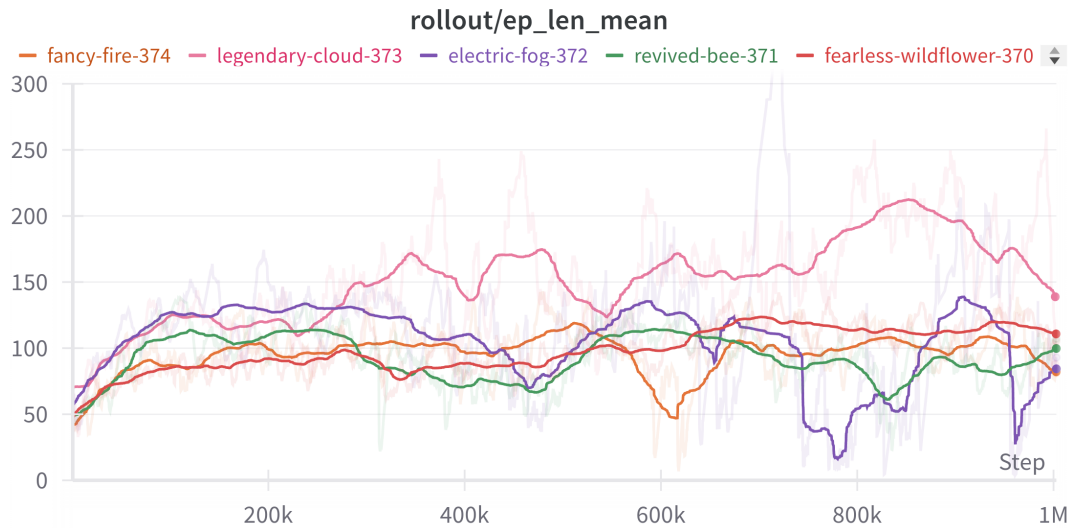


Figure 4.4: Smoothed episode length for trial with shoulder control

Name	Gamma	Learning rate
fancy-fire-374	0.99	0.0003
legendary-cloud-373	0.999	0.003
electric-fog-372	0.999	0.001
revived-bee-371	0.99	0.001
fearless-wildflower-370	0.99	0.003

Table 4.5: Model hyperparameters for 4.4.

Only legendary cloud managed to consistently balance for longer than its counterparts. Notably it took longer too reach a level similar to faithful universe and clear dragon in the previous experiment, likely due to the fact that the action space was expanded. It did however manage to stay dominant until the end of the training cycle. It's parameters were therefore propagated to the next round of experimentation.

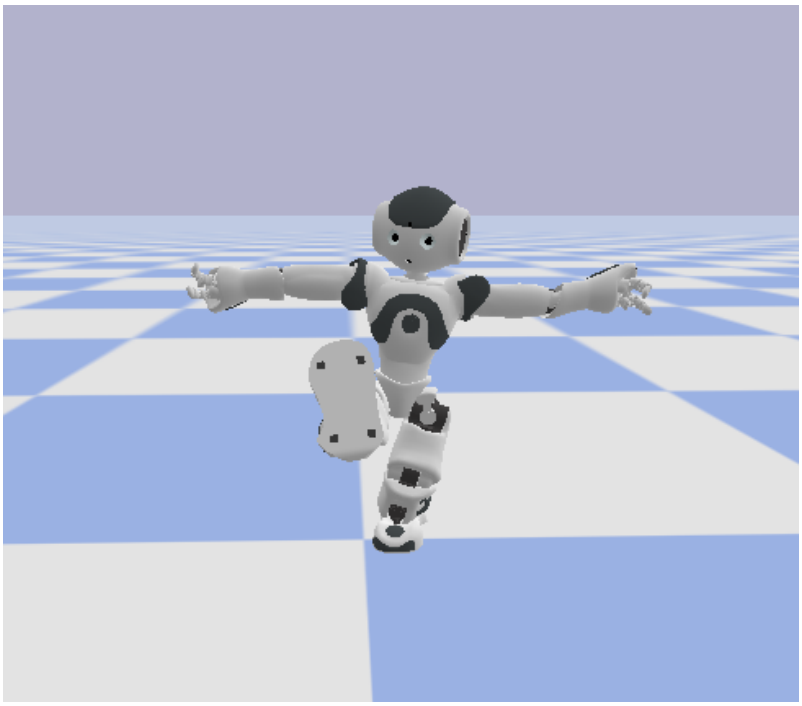


Figure 4.5: The Nao robot lifting its foot during training.

Some models were observed to move their foot unnecessarily high, one such instance can be seen in figure 4.5. To combat this a change was made to the foot height reward to decrease the reward as the foot moves above the target height. The details of this change are outlined in the method 4.3.

4.3.3 Final results

The batch and model size were experimented with next, as well as the rewards for having the COM above the support polygon and keeping the supporting foot on the ground.

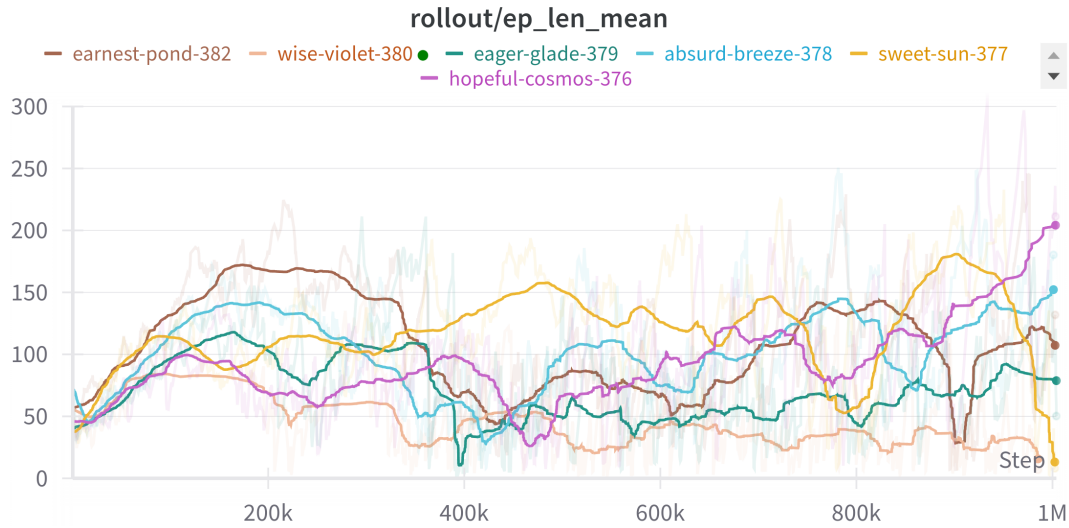


Figure 4.6: Smoothed episode lengths for the last experiment.

Name	COM above support	Contact points	Batch size	Model size
earnest-pond-382	1	1	512	[512, 512]
wise-violet-380	1	1	1028	[256, 256]
eager-glade-379	1	0	256	[256, 256]
absurd-breeze	0	1	256	[256, 256]
sweet-sun-377	1	1	256	[256, 256, 256]
hopeful-cosmos-376	1	1	256	[512, 512]

Table 4.6: Reward function parameters and model hyperparameters for figure 4.6

Eager glade performed significantly worse than absurd breeze, indicating that the contact point reward is more important for the learning process. This makes sense as there already is a reward to incentives moving the COM above the support polygon, namely the distance penalty.

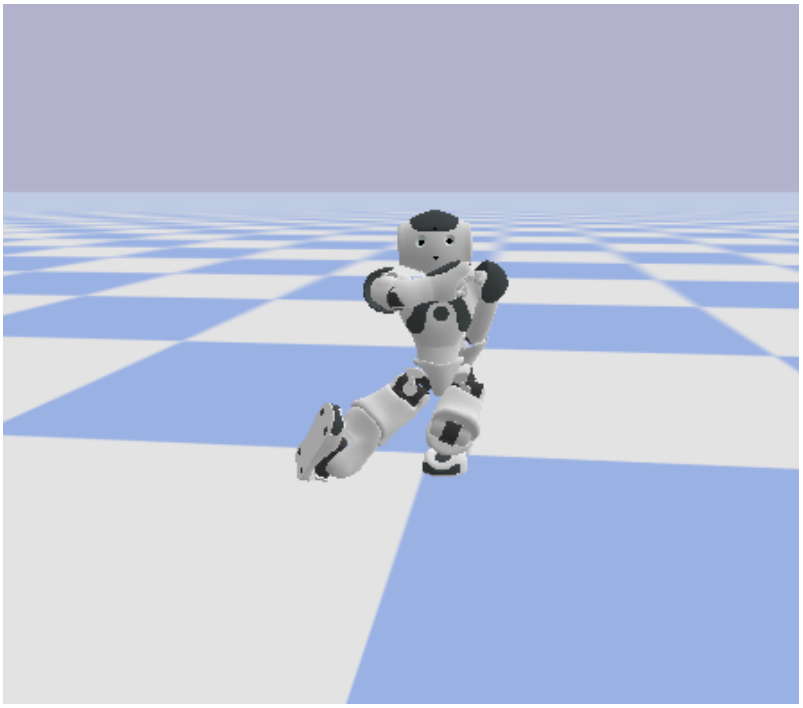


Figure 4.7: The last stance learned by legendary cloud.

The final stance learned by hopeful cosmos can be seen in the figure above, note that this is not a completely stable and can only be maintained for around 4 seconds before falling over. Besides this it is not an ideal stance to perform the rest of a kicking motion from. The model has learned to lower its COM, this could also be observed for other models in this experimentation phase. Some examples of attempted stances are provided in appendix section 7.2.

Chapter 5

Conclusion

In this study, the potential of the Soft Actor Critic algorithm to achieve a stable one legged stance was investigated. The research aimed to create a purely reinforcement learning based model for dynamic balancing to lay a foundation for moving away from the static kick that is currently deployed by the Dutch Nao team.

A small initial test was performed to provide confidence in SAC as choice for the dynamic balancing task, namely using the cartpole task. This yielded reaffirming results.

The experiments were conducted in a simulated environment instead of with a physical robot to facilitate more rapid prototyping. By adjusting various hyperparameters and reward function components, it was possible to identify configurations that led to improved stability and performance.

Despite the progress made, several technical challenges were encountered. The discrepancies between the simulated and physical capabilities of the robots highlighted the limitations of the simulation environment. Additionally, the initial implementation of qiBullet sensor readings caused significant delays.

The time constraints caused by the technical challenges limited the extent of experimentation and optimization possible within the scope of this research. Future work could focus on performing more extensive hyperparameter searches and refining of the reward function.

In conclusion, this study lays a foundation for utilizing reinforcement learning algorithms, particularly SAC, in enhancing the dynamic capabilities of humanoid robots. The findings suggest that with further refinement and exploration, it is feasible to develop more sophisticated and adaptable balancing behaviors, paving the way for improved performance in competitive robotics scenarios like RoboCup.

Chapter 6

Discussion

6.1 Technical challenges

6.1.1 Simulator

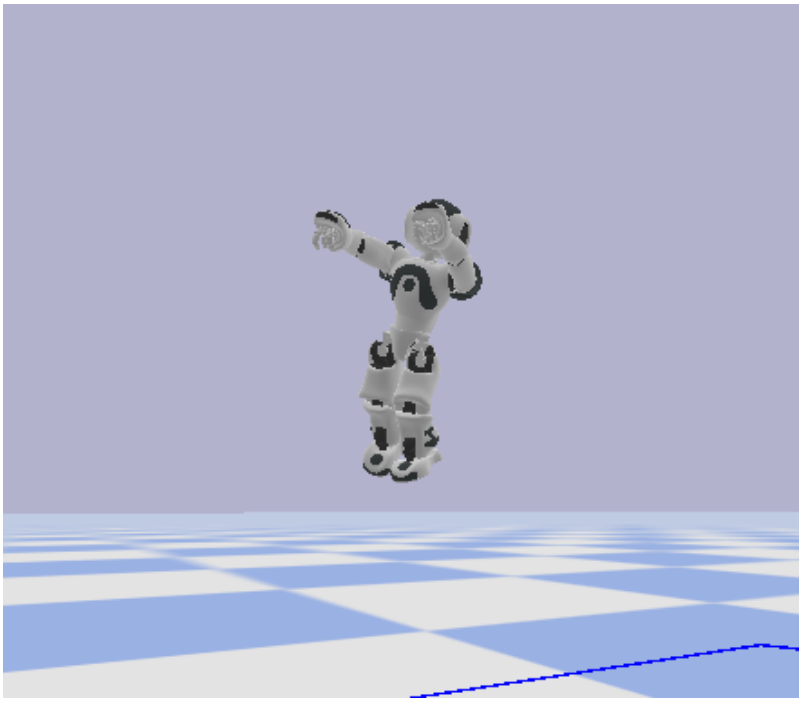


Figure 6.1: The Nao robot jumping during training

During training it could be observed that the robot jump up into the air while keeping upright. It achieves a jump of approximately 50cm, which is almost its

own height. The physical robot, even with the help of an energy storing elastic device, cannot achieve a jump of over 14mm (Hondo, Kinase, & Mizuuchi, 2012). That is over 30 times less than observed in the simulator. This shows a clear discrepancy in the capabilities of the physical and simulated robot in terms of strength. This observation was the main reason for introducing the failing condition for the supporting leg leaving the ground and reducing the maximum joint velocity to 10%.

6.1.2 qiBullet sensor implementation

It was observed that the simulations for the nao robot experiments initially ran at a lower than expected pace, taking multiple hours to run approximately 35000 time steps. This would significantly reduce the amount of experimenting that could be done, as a single experiment is 500000-1000000 time steps. Upon further investigation it was found that the culprit was the implementation of qiBullets sensor readings. The sensors were using a threading lock. The qiBullet documentation does not elaborate on the reason for this choice [1]. Since the IMU sensors are located in the torso of the robot, a function was made to read the angular and linear velocity of the torso link instead. A version of the FSR reading sensors was also created that does not rely on the Sensor class by qiBullet. Both can be found in appendix section 7.1.

6.2 Time limitations

Due to programming mistakes resulting in the redoing of the experiments in a late stage of the research for the Nao robot balancing, task the amount of time was available for testing was severely reduced. It is in the nature of reinforcement learning that gathering results and iterating upon them takes a considerable amount of time as the training process has to be repeated for every change. Given more time a more extensive search through the optimise parameters would have been feasible. Additionally there are many ways to improve and expand upon the the research performed during the writing of this paper, as will be discussed in the section below.

[1] `https://softbankrobotics-research.github.io/qibullet/classqibullet_1_1imu_1_1Imu.html#a3699aac415978b7b06aa72fecc2a8f13`

6.3 Future work

The limitations encountered in this study highlight the need for further research to conduct a more thorough search for the ideal reward parameters and model hyperparameters. Several avenues for future exploration additionally include:

Incorporating standing on one leg into a full kicking motion:
The final stance learned was not ideal for performing the kicking motion, this may change once it is trained in the context of the full task. **Different control strategies:**
Instead of velocity control position or torque control could be investigated instead
Adjusting from Static Kicks:
Starting from a static kick and adjusting key points, could provide a more stable foundation for dynamic balancing. This is a hybrid approach and reduces the action space to search through.
Exploring Related Work Approaches:
Implementing and testing the various methods discussed in the related work section to identify potential improvements and alternative strategies.
Proximal Policy Optimization (PPO):
A reinforcement learning algorithm like PPO may be more suitable for the balancing task. **Parameter Consistency:**
Running models with the same parameters multiple times to obtain standard deviations and see clearer differences in performance, thereby ensuring more robust conclusions.
Transfer to Physical Robot:
Bridging the gap between simulation and real-world application by transferring the learned models to physical Nao robots and validating their performance in actual environments.

References

Abreu, M., Silva, T., Teixeira, H., Reis, L. P., & Lau, N. (2021). 6d localization and kicking for humanoid robotic soccer. *Journal of Intelligent & Robotic Systems*, *102*(2).

Aldebran Robotics. (n.d.). *qibullet.* Retrieved from `https://www.aldebaran.com/developer-center/articles/Qibullet_simulator/index.html` (Accessed: 12-06-2024)

Bong, J. H., Jung, S., Kim, J., & Park, S. (2022). Standing balance control of a bipedal robot based on behavior cloning. *Biomimetics*, *7*(4), 232.

Busy, M., & Caniot, M. (n.d.). qibullet, a bullet-based simulator for the pepper and nao robots. Retrieved from `https://arxiv.org/pdf/1909.00779`

Draskovic, M. (2022). Design of a dynamic omnidirectional kick engine for nao bipedal robots in robocup. Retrieved from `https://robocup.ethz.ch/wp-content/uploads/report_marina_draskovic.pdf`

Dulac-Arnold, G., Levine, N., Mankowitz, D. J., Li, J., Paduraru, C., Gowal, S., & Hester, T. (2021). Challenges of real-world reinforcement learning: definitions, benchmarks and analysis. *Machine Learning*, *110*(9), 2419–2468.

Engelke, T. (2021). Learning to kick from demonstration with deep reinforcement learning. Retrieved from `https://www.researchgate.net/publication/357118740_Learning_to_Kick_from_Demonstration_with_Deep_Reinforcement_Learning`

Erez, T., & Smart, W. D. (2008). What does shaping mean for computational reinforcement learning? In *2008 7th ieee international conference on development and learning.* doi: 10.1109/DEVLRN.2008.4640832

Haarnoja, T., Zhou, A., Abbeel, P., & Levine, S. (2018). Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. *CoRR*, *abs/1801.01290*. Retrieved from `http://arxiv.org/abs/1801.01290`

Hondo, T., Kinase, Y., & Mizuuchi, I. (2012). Jumping motion experiments on a nao robot with elastic devices. In *2012 12th ieee-ras international conference on humanoid robots (humanoids 2012)* (p. 823-828). doi: 10.1109/HUMANOIDS.2012.6651615

Nakada, M., Allen, B., Morishima, S., & Terzopoulos, D. (2010). Learning arm motion strategies for balance recovery of humanoid robots. *2010 International Conference on Emerging Security Technologies*. doi: 10.1109/EST.2010.18

Rezaeipanah, A., Amiri, P., & Jafari, S. (2021, 09). Performing the kick during walking for robocup 3d soccer simulation league using reinforcement learning algorithm. *International Journal of Social Robotics*, *13*. doi: 10.1007/s12369-020-00712-2

Chapter 7

appendix

7.1 Sensor code

This code was heavily inspired by the original implementation by United Robotics in the qiBullet simulator and falls under the Apache license 2.0 [1]

```
import pybullet as p

class Sensor:
    def __init__(self, robot, client):
        self.robot = robot
        self.client = client

class Imu(Sensor):
    def __init__(self, robot, imu_link, client):
        super().__init__(robot, client)
        self.imu_link = imu_link
        self.angular_velocity = [0.0, 0.0, 0.0]
        self.linear_velocity = [0.0, 0.0, 0.0]
        self.linear_acceleration = [0.0, 0.0, 0.0]
        self.last_time = 0.0

    def get_imu_data(self):
        try:
            link_state = p.getLinkState(
                self.robot,
                self.imu_link.get_index(),
```

[1] https://github.com/softbankrobotics-research/qibullet/blob/master/LICENSE

```
                computeLinkVelocity=True)
            current_time = time.time()
            dt = current_time - self.last_time

            self.angular_velocity = link_state[7]
            self.linear_acceleration = [
                                (i - j) / dt for i, j in zip(
                                    link_state[6],
                                    self.linear_velocity)]
            self.linear_velocity = link_state[6]
            self.last_time = current_time

            return self.angular_velocity, self.linear_velocity, self.linear_accele
        except Exception as e:
            print("warn:", e)
            return self.angular_velocity, self.linear_velocity, self.linear_accele



class Fsr(Sensor):
    LFOOT = "l_ankle"
    RFOOT = "r_ankle"

    def __init__(self, robot, fsr_link, client, gravity = -9.81):
        super().__init__(robot, client)
        self.fsr_link = fsr_link
        self.gravity = gravity

    def get_force(self):
        contact_tuple = p.getContactPoints(
            bodyA=self.robot,
            linkIndexA=self.fsr_link.get_index())

        if not contact_tuple:
            return [0.0, 0.0, 0.0, 0.0]

        return [contact[9] for contact in contact_tuple]
```

7.2 Learned stances in the final round of experimentation

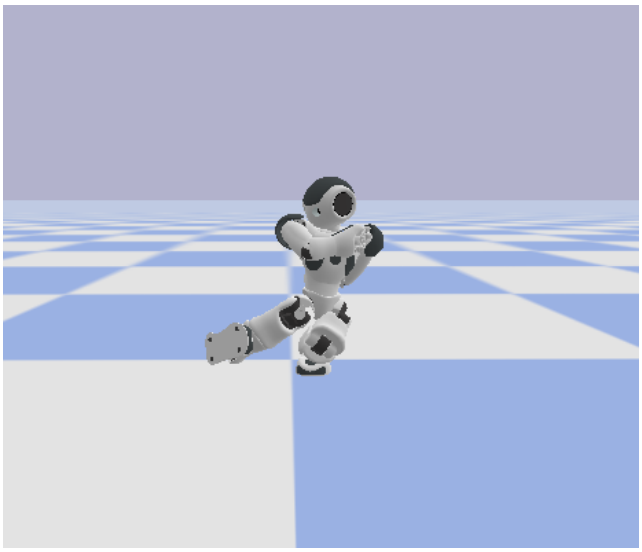


Figure 7.1: Learned stance 1.

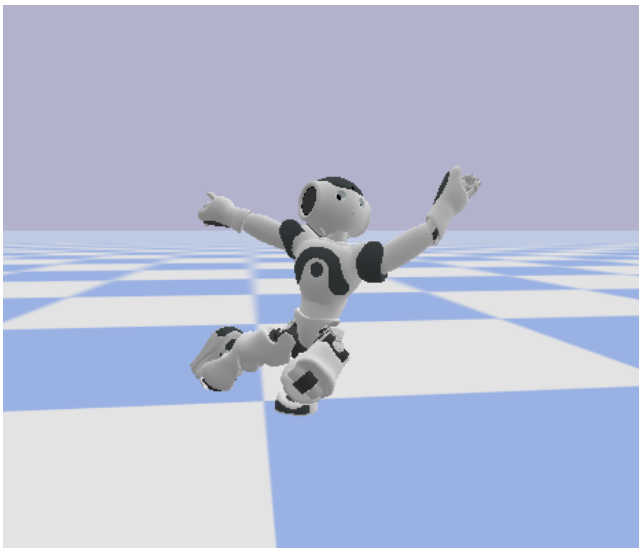


Figure 7.2: Learned stance 2.

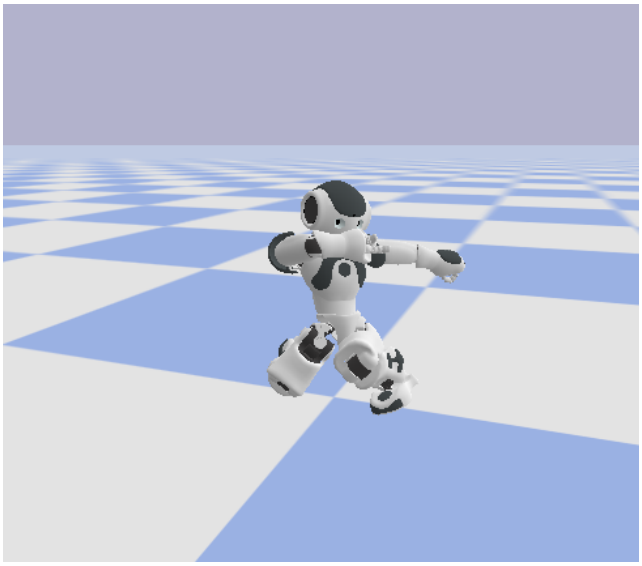


Figure 7.3: Learned stance 3.