UNIVERSITY OF BIRMINGHAM

Optimising Swing Motion of a Nao Robot on a Hinged Swing

Robotics Group Study 2019

D. Apetroaei, M. Avery, T. Brent, J. Doering, B. Hughes, A. Jermakov, F. Leung, C. Li, J. Matthews, K. Mogilner, A. Nicholls, G. Sheppard, W. Sullivan, D. Thomas, H. Tucker

Abstract

This study aimed to apply the principles of human swinging - starting, swinging with minimal exertion, for maximum amplitude, and stopping independently - to a Nao robot. These aims were approached in three ways, through the use of theory and simulation, creating code to directly apply to the robot, and developing machine learning algorithms. A hinged flail model was simulated to test different pumping techniques, both parametric and rotational motion were compared - finding the maximum amplitudes reached were 84.7° and 27.4° respectively. Energy conservation was applied to evaluate the theoretical maximum angle the swing could reach, given different initial release parameters. Self-start methods were investigated, with Nao pumping the swing at the system's fundamental frequency considered the most promising method. Fundamental angular frequencies for different models were found, falling within the range $(2.48 \pm 0.01) rads^{-1}$ to $(2.56 \pm 0.01) rads^{-1}$. Electrical efficiency of Nao was found to be $11.56 \pm 0.92\%$. Mechanical efficiency was investigated for Nao, as well as for theory and machine learning simulations. The maximum mechanical efficiency of Nao was $(9.5 \pm 0.2)\%$. Several algorithms were developed, allowing complete control over the amplitude Nao could reach and maintain. These included a self-start script, rotational motion (which reached a maximum swing amplitude of 14^o) and parametric motion (which reached a maximum amplitude of 27^o). The speed at which Nao switched between different postures was improved, with a 33% increase compared to 2018. The optimal point in a swing cycle for Nao to kick, that would give the sharpest increase in amplitude, was investigated. This was found to be 0.25s before the peak of the swing. Four different machine learning algorithms were developed: Q-Learning, Backwards Q-Learning, Evolutionary Neural Networks and Proximal Policy Optimisation. Q-Learning and Backwards Q-Learning techniques were both successfully applied to rotational swinging. Evolutionary Neural Nets were implemented, for solely parametric, and combined parametric and rotational pumping; it was concluded that this was the most auspicious machine learning algorithm. Proximal Policy Optimisation was used to develop a three-dimensional learning environment. Following testing on a simulation, the evolutionary neural nets and Backward Q-Learning were applied to Nao.

Keywords: Nao, Robotics, Machine learning, Lagrangian mechanics

University of Birmingham
Physics and Astronomy Department

Contents

1 Introduction to Machine Learning Sub-Group

Fiesta Leung

1.1 Introduction

Millions of years of evolution have given the human species the ability to adapt to the surrounding environment. Humans are capable of behaving appropriately to unforeseen situations with instincts and reflexes based on past experiences. Despite the complex connectivity between neurons inside the human brain, its capacity to store every single piece of information received is limited. Computers, by contrast, do not possess the intelligence of humans yet they have the ability to store a considerably larger amount of data. This means that if computers can be programmed to exhibit human-like intelligence, they have the potential to outperform the human brain.

From basic arithmetic to the triangulation used in GPS, computers have been programmed to perform tasks of ever-increasing complexity over the last several decades. The emergence of databases and the gradual advancement in computer performance means that not only can our lives be converted and stored as digital data for learning and analysis purposes, but tasks which require intelligence can be learnt and carried out by computers. The yearning for artificial intelligence has therefore spawned the quest for machine learning.

1.2 Background Theory

Machine learning is a method of data analysis through which a computer system optimises its performance progressively, by learning the underlying structure to generate a model to fit the corresponding data without being programmed explicitly: a model which is improved constantly as more data is collected. It therefore plays an important role in achieving an autonomous world.

Machine learning can be sub-categorised into supervised, unsupervised and reinforcement learning, listed in order of increasing difficulty of implementation. In supervised learning, the computer is fed explicitly-defined input-and-output pairs. It is advantageous in the sense that the decision boundary can be defined as specifically as one desires, meaning the predictions made by the algorithm are of high accuracy given the input is of one of those pairs learnt during training. This type of machine learning, however, is limited by its flexibility. In other words, the accuracy of the output requires the corresponding input to be fed during training. In contrast, inputs given in unsupervised learning do not have the corresponding predefined outputs. The algorithm therefore, has the freedom to learn and explore the hidden patterns in the given data set and predict outputs based on the model learnt during training. Although outputs generated might be of lower accuracy, the discovery of new knowledge of a data set through exploration makes it a desirable method of training computers in reaching artificial intelligence. Unlike the previous two techniques, reinforcement learning is a trial-and-error learning method where an algorithm learns about an environment with predefined action and state spaces via continuous interaction. Rewards are then given to help the algorithm improve its performance. Hence, the longer the training, the more information on its environment an algorithm accumulates and subsequently the more optimal is its performance in achieving its predefined goal.

1.3 Related Work

[1] discusses machine learning from a historical viewpoint through the example of Alan Turing's imitation game. The article runs through the fundamental principles of reinforcement learning and addresses its

importance and challenges in the field of machine learning in establishing artificial intelligence. Machine learning has proved its potential in advancing autonomous robotics application. [2] presents how the champions of the 2011 RoboCup simulation league successfully implemented machine learning, to optimise the walk engine parameter values to achieve faster walking on the NAO robot. Reinforcement learning in particular, has demonstrated its efficacy in robotics through improving the adaptability of soft robots to an environment in [3].

Studies have been carried out to evaluate the performances of different machine learning algorithms. [4] looks into different ways of evaluating the knowledge learnt by the agent, through the Q-learning algorithm. For instance, the learning-rate parameter used in updating the Q-values was altered to compare the corresponding convergence rates. In additions, [5] compares the performances of several reinforcement learning algorithms in optimising the motion of the controller in the Cart-pole problem. These algorithms include Q-Learning, value function approximation and actor-critic. Results show that actor-critic has faster convergence rate than Q-learning in discrete state spaces and value function approximation demonstrates top performance between the three.

Amongst the three types of machine learning, reinforcement learning had been chosen by the robotics teams from 2016, 2017 and 2018 to teach the humanoid robot NAO to perform human swinging motions. [6] applied Q-Learning to the inverted pendulum problem. The learning was unsuccessful, resulting in the algorithm not being tested on the NAO robot. The robotics team from 2017 extended the dumbbell pendulum model in modeling the swinging system. Three learning algorithms including Actor-Critic, Q-Learning and SARSA were tested. The approach of neural networks was also considered. None of these algorithms however, were implemented on the robot NAO [7]. Furthermore, the team from 2018 looked into various reinforcement learning algorithms and concluded Q-Learning to be the most effective algorithm in optimising the swinging motion of NAO [8].

2 Backward Q-Learning based SARSA

2.1 Justification

Fiesta Leung

Despite its effectiveness in converging to an optimal policy, Q-Learning suffers from a slow convergence rate as mentioned in section 17.1. This is due to the fact that the precision of an optimal Q-value function requires the learning agent to visit the state-space infinitely many times [9]. In other words, a large number of trials is required for the converged optimal policy to be precise. This trade-off between the precision of the optimal policy and the training time of the agent accounts for the slow convergence rate of Q-learning, imposing constraints on applying Q-Learning on the NAO robot as the machine learning subgroup in 2016 noted [6]. In hindsight, it would be ideal to develop a Q-Learning based algorithm, of faster convergence, to circumvent the problem of overheating the joints of NAO due to the long training time. The successful development of such algorithm will not only have a greater potential to train the NAO robot to swing in an optimal manner, but more importantly, will address the issue of the slow convergence of Q-Learning in finely-discretised environments. This is of great importance as the convergence problem becomes exacerbated due to the environment becoming more complex in general.

In fact, a special Q-Learning algorithm, known as Backward Q-Learning (BQL), was proposed by Wang, Li and Lin (2013) to serve as the basis of several Q-Learning based algorithms [10]. Such algorithms are combinations of BQL integrated with various reinforcement learning (RL) algorithms. These RL algorithms include but are not limited to SARSA by Sutton and Barto [11], adaptive Q-Learning by Hwang et al. [12] and Simulated Annealing (SA)-Q-Learning by Guo et al. [13]. Comparisons were made between the performances of these Q-Learning based algorithms and those of the corresponding RL algorithms on three machine learning problems, namely the cliff-walking, the mountain car and the cart-pole balancing problems [10]. The outperformance of the Q-Learning based algorithms over the corresponding RL algorithms in all three problems demonstrated the effectiveness of combining BQL and individual RL algorithms in enhancing convergence rate and improving final performance. Performance comparsions for the cart-pole balancing problem were of particular interest as the nature of this problem is the closest to the swinging problem concerned in this project.

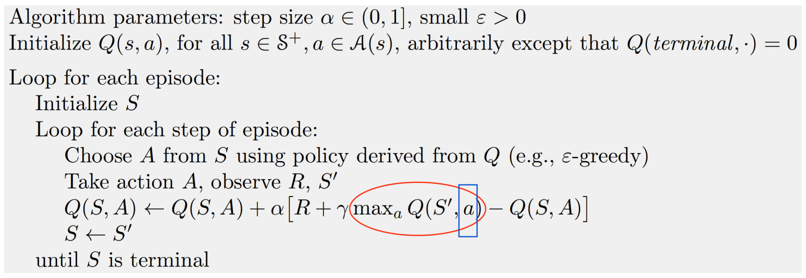
Although the BQL based SA-Q-Learning was concluded to be the best-performing algorithm for the cart-pole balancing problem, the BQL based SARSA algorithm (BQSA) was chosen instead in approaching the swinging problem. This was because the complexity in coding the SA-Q-Learning algorithm did not justisfy the edge it has over SARSA given the time frame for this project.

2.2 Q-Learning vs SARSA

Fiesta Leung

Similar to Q-Learning, SARSA is another model-free reinforcement learning algorithm. It was introduced by Rummery and Niranjan as Modified Connectionist Q-Learning (MCQ-L) in 1994 as a modified version of the original Q-learning algorithm by Watkins [14], in the hope of extending the size of the state-space examined by a learning agent without lowering its efficiency to converge to an optimal policy [15]. As this modified algorithm updates its value function with every element of the quintuple of events ($\mathbf{S}tate_t, \mathbf{A}ction_t, \mathbf{R}eward_{t+1}, \mathbf{S}tate_{t+1}, \mathbf{A}ction_{t+1}$) that comprises the transition from a state-action pair to another [15], it is commonly known by its alternative name SARSA.

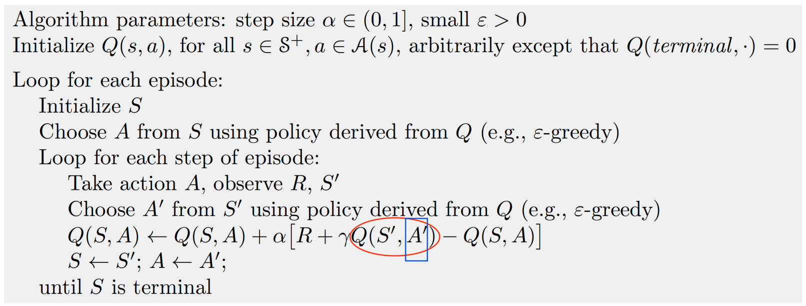
Both Q-Learning and SARSA are model-free RL algorithms which employ the method of Temporal-Difference (TD) learning in learning their corresponding value function [11]. In essence, TD learning allows learning algorithms to learn the dynamics of the given environments through direct experiences and to update their estimated models of the environments partially based on previous experiences after every timestep [11]. The two learning algorithms differ in the sense that Q-Learning is an off-policy learning technique while SARSA is on-policy. Figures 2.1 and 2.2 below outline the frameworks of Q-Learning and SARSA respectively.



```
Algorithm parameters: step size $\alpha \in (0,1]$, small $\varepsilon > 0$
Initialize $Q(s,a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize $S$
  Loop for each step of episode:
    Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
    Take action $A$, observe $R$, $S'$
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S', \boxed{a}) - Q(S,A)\big]$
    $S \leftarrow S'$
  until $S$ is terminal
```

Figure 2.1: Framework of the Q-Learning algorithm (where α and γ are the learning rate and discount factor respectively). Taken from [11].



```
Algorithm parameters: step size $\alpha \in (0,1]$, small $\varepsilon > 0$
Initialize $Q(s,a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize $S$
  Choose $A$ from $S$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
  Loop for each step of episode:
    Take action $A$, observe $R$, $S'$
    Choose $A'$ from $S'$ using policy derived from $Q$ (e.g., $\varepsilon$-greedy)
    $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma Q(S', \boxed{A'}) - Q(S,A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until $S$ is terminal
```

Figure 2.2: Framework of the SARSA algorithm (where α and γ are the learning rate and discount factor respectively). Taken from [11].

The difference between the two algorithms lies in the corresponding updating formula for their own Q-value function, as marked in figures 2.1 and 2.2. In Q-Learning, the expected future reward (circled in red in figure 2.1) used to update the Q-value function $Q(S, A)$ is maximised. It is calculated using (1) the next state S' the agent enters after taking action A in the current state S and (2) the action a available in the next state S' (marked in blue in figure 2.1) that gives the highest Q-value. This means the algorithm updates the Q-value function assuming a greedy policy was being followed despite the fact that the agent may be following an exploratory policy(i.e. not acting greedily). Q-Learning is therefore an off-policy learning technique as the Q-value function is not updated based on the current policy followed by the agent.

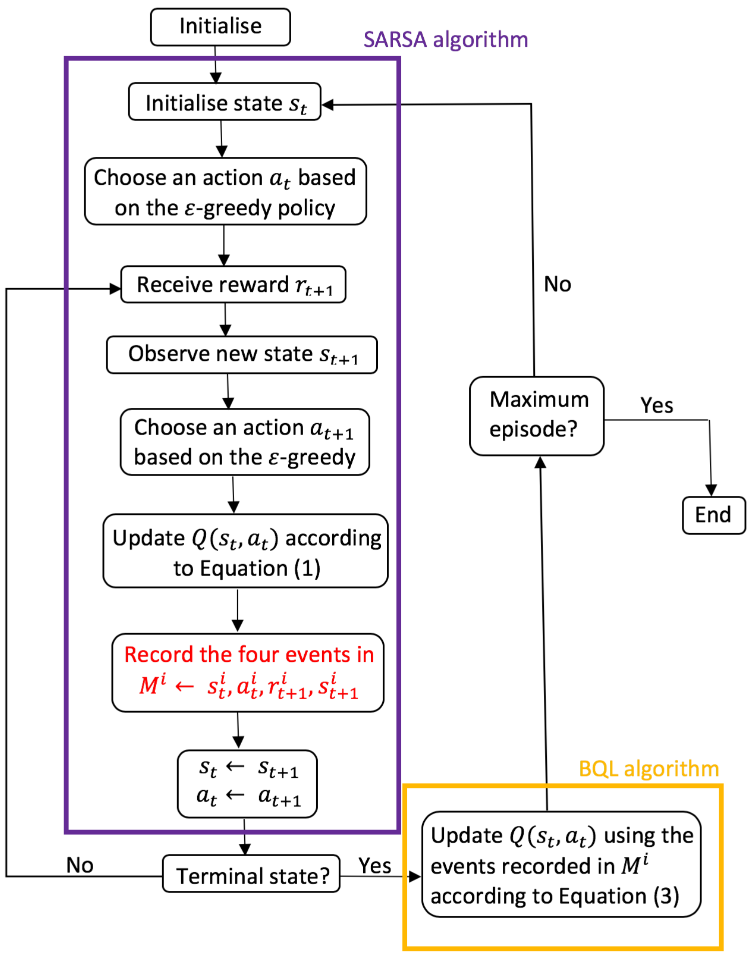
In contrast, SARSA is an on-policy learning algorithm; as the expected future reward (circled in red in figure 2.2) is calculated using (1) the next state S' the agent enters after taking action A in the current state S and (2) the next action A' (marked in blue in figure 2.2) chosen from the next state S' using the current policy followed by the agent. The current state S and the current action A are then updated to the next state S' and the next chosen action A' respectively, followed by the chosen action A' being executed in the next timestep to observe the subsequent reward and state. As the current policy followed by the agent is being evaluated during training, SARSA results in a faster convergence rate than Q-Learning [16]. Q-Learning however, has a better final performance in general despite its slow convergence rate [10]. This has been demonstrated in solving the cliffwalk problem [17].

Ideally, one would want to design an algorithm which exhibits the fast converging characteristic of SARSA and the optimal final performance of Q-Learning. By integrating BQL with SARSA, BQSA has been proven to embody the fast converging characteristic of SARSA while maintaining the quality of the final performance achieved by Q-Learning [10].

2.3 Theory

Fiesta Leung

BQSA, as mentioned in the section above, is the combination of BQL and SARSA. The flowchart of BQSA is presented in figure 2.3 below [10].



Initialise
SARSA algorithm
Initialise state s_t
Choose an action a_t based on the ε-greedy policy
Receive reward r_{t+1}
Observe new state s_{t+1}
Choose an action a_{t+1} based on the ε-greedy
Update $Q(s_t, a_t)$ according to Equation (1)
Record the four events in $M^i \leftarrow s_t^i, a_t^i, r_{t+1}^i, s_{t+1}^i$
$s_t \leftarrow s_{t+1}$ $a_t \leftarrow a_{t+1}$
Terminal state?
No
Yes
BQL algorithm
Update $Q(s_t, a_t)$ using the events recorded in M^i according to Equation (3)
Maximum episode?
No
Yes
End

Figure 2.3: Flowchart of BQSA

As seen from the flowchart in figure 2.3, the construction of BQSA can be broken down into two main components as indicated in the figure; the SARSA and the BQL algorithms. The SARSA algorithm is the fundamental building block of BQSA. It is no different to the ordinary SARSA algorithm except there is an extra step of recording four events in each timestep of an episode (as marked in red in the flowchart) in a matrix M^i. These four events include:(1) the current state s_t^i, (2) the action a_t^i chosen from state s_t^i using the policy (i.e. ε-greedy policy) followed by the agent, (3) the immediate reward r_{t+1}^i received for executing action a_t^i in state s_t^i and (4) the new state s_{t+1}^i observed (where $i = 1, 2, 3, ..., N$ denotes the i^{th} time the Q-value function $Q(s_t^i, a_t^i)$ is being updated in the current episode). Nevertheless, the BQL algorithm remains the main feature of BQSA.

The BQL algorithm essentially uses the same updating formula for the Q-value function as the standard Q-Learning algorithm discussed in section 17.1. It however, updates the Q-value function in a reversed order using the set of quadruple events recorded from the RL algorithm it is integrated with. In BQSA, the BQL algorithm 'backward' updates the Q-value function using the set of quadruple events recorded from the SARSA algorithm once the terminal state is reached in the current episode.

To illustrate this, consider the following updating equation of SARSA [10]:

$$Q(s_t^i, a_t^i) \leftarrow Q(s_t^i, a_t^i) + \alpha(r_{t+1}^i + \gamma Q(s_{t+1}^i, a_{t+1}^i) - Q(s_t^i, a_t^i)) \tag{1}$$

During each timestep in an episode, the Q-value function will be updated according to equation 1 while the quadruple of events $(s_t^i, a_t^i, r_{t+1}^i, s_{t+1}^i)$ are being simultaneously recorded in a matrix M^i as follows [10]:

$$M^i \leftarrow s_t^i, a_t^i, r_{t+1}^i, s_{t+1}^i \tag{2}$$

.

Once the terminal state is reached in the current episode, the BQL algorithm will utilise the set of quadruple of events recorded in M^i to update the Q-value function N times in a reverse manner according to the following equation [10]:

$$Q(s_t^j, a_t^j) \leftarrow Q(s_t^j, a_t^j) + \alpha_b(r_{t+1}^j + \gamma_b maxQ(s_{t+1}^j, a) - Q(s_t^j, a_t^j)) \tag{3}$$

where $j = N, N-1, N-2, ..., 1$, α_b and γ_b are the learning rate and discount factor used in the BQL algorithm respectively.

The method of backward updating does in fact mimick the process of "reverse replay" carried out by hippocampal neurons called place cells. Reverse replay is a type of experience replay where an experience is replayed in the opposite direction as experienced. Studies on the relation between the change in rewards and the direction of experience replay have shown that reverse replay is more effective than forward replay in reward-based learning [18] and is likely to play a part in optimising goal-directed paths [19]. BQSA was therefore investigated in the hope of optimising the swinging motions of the humanoid robot NAO as this was a reward-based learning problem.

The main advantage of updating backwards (i.e. from goal to start) is that, unlike the standard forward updating (i.e. from start to goal) in Q-Learning, it ensures every state visited by the learning agent has a path leading to the goal state. This in turn enables a more precise estimation of the Q-value function for the vicinity of the goal state as exploring the environment in reverse allows the agent to learn how states will change prior to the goal state. Backward updating becomes even more desirable when solving sparse-reward RL problems since the rewards assigned to near-goal-states are often considerably higher, motivating the agent to achieve the goal. Learning backwards from the goal state therefore allows the agent to gain more knowledge on the reward function, reducing the training time.

2.4 Epsilon-greedy policy

Fiesta Leung

The aim of a RL agent is to select actions which maximise the expected cumulative reward in reaching the goal state. To date, maintaining the balance between exploration and exploitation remains one of the main challenges in achieving this goal. This dilemma arises from the fact that the agent can never solely rely on pure exploration or exploitation. Exploitation ensures the agent learns from and employs its previous experiences which are beneficial to the accumulation of rewards. Yet, exploration is crucial in maximising the cumulative reward as pure exploitation may cause unexplored actions which potentially yield greater rewards to be disregarded. Additionally, exploration is often essential at the beginning as the agent has no knowledge of the environment of which it can exploit. The action-selection policy evaluated by the agent therefore plays an important role in assisting the agent to optimise its performance [20].

The action-selection policy chosen for the BQSA algorithm was the ε-greedy policy. It is a strategy which decides whether the agent explores or exploits in a given state [21]. The ε denotes the probability of a random action being chosen in a given state (i.e. to explore) while $1 - \varepsilon$ represents the probability of choosing the action with the maximum expected reward in a given state (i.e. to exploit). If $\varepsilon = 1$, the agent will only carry out exploration. Conversely, $\varepsilon = 0$ signifies a greedy policy being followed. This means the agent will always pick the action associated with the highest reward. The framework of the ε-greedy policy is outlined in figure 2.4 below [21].

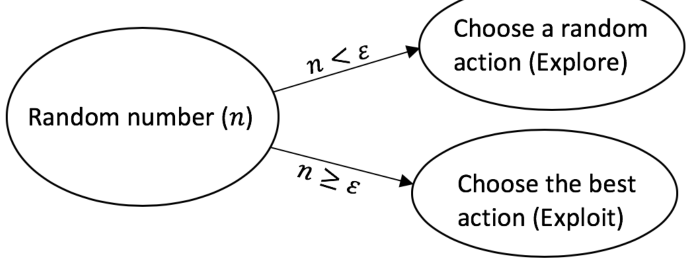

Random number (n)
$n < \varepsilon$
Choose a random action (Explore)
$n \geq \varepsilon$
Choose the best action (Exploit)

Figure 2.4: Framework of the ε-greedy policy.

In order for the algorithm to execute the ε-greedy policy, a random number n is assigned to each action selection. As illustrated in Figure 2.4, if $n < \varepsilon$, the agent will explore and exploit otherwise. In approaching the swinging problem, the ε was formulated as a value which decreases at a constant rate as the training progresses. It was assigned the value of 1 at the beginning since the agent would have no knowledge on either the environment or the set of actions available. Any exploitation at this stage is likely to lengthen the converging time without being advantageous to the agent's learning process. It is however beneficial for the agent to increasingly take advantage of the knowledge it accumulates about the environment as the training advances. As exploration and exploitation are mutually exclusive, the extent to which the agent explores has to be reduced to allow more exploitations to take place during the action-selection process. Hence, the ε value was coded to decay at a constant rate over time.

Denisa Apetroaei

2.4.1 Q-Table

The Q-table in Backward-Q-Learning was structured as a 2X4 table for each rotational and parametric swinging. There are two actions possible for each one of them:extended/seated for rotational and raised/lowered for parametric.
The four states list elements are defined depending on the area where the swing is:positive angle with positive velocity, positive angle with negative velocity, negative angle with negative velocity,negative angle with positive velocity.
The reward is given as the change in the total energy of the system. This way if an action that would decrease energy is taken, the reward would be negative so the robot would avoid that action for that specific state.

2.4.2 Code Implementation

A class *QLearn* is created for the Backwards Q algorithm with each step from Fig.2.3 being defined as a different method[22]. Here the *__init__* method creates the blank Q table depending on the action-space matrix.The actions are different for rotational and parametric, but the states are defined the same.
The *getQ* method returns the current element of the Q-table for the given state and action. Function *learnQ* updates element depending on the reward given. The epsilon-greedy policy is applied in the method *chooseAction*.
These methods are called in the *learn_Ql* and *learn_SARSA* methods which apply the Bellman equation[14]

equivalent for each algorithm, Q-learning and SARSA.
A different script class calls the *Qlearn* object, defines states for the space elements of the Q-table and updates it by calling the object's methods according to the steps illustrated.
The current state and energy are obtained through the functions *get_state* and *get_energy* which use the position,velocity and energy parameters from the simulation as arguments. These represent the inputs of the algorithm together with the action choice based on the Epsilon-Greedy policy [9].

2.5 Results

2.5.1 Choice of constants

Repeated runs were made with different parameters. By varying them, the angle achieved was compared to the episode number as a representative of time.
The main three constants are α and γ from Eq.3 together with the ϵ from the Epsilon-Greedy policy. The ϵ from the Epsilon-Greedy policy determines if a random action is taken. If the action's probability is lower than epsilon, it passes, otherwise a new action is chosen based on the highest value of the Q-table for the current state. This determines a high ϵ to encourage exploration, by increasing the probability of random actions being considered valid, while a small epsilon encourages exploitation by increasing the probability of using actions which effects on the system are known.
It is expected that the values close to 1 to never train, as the system is nothing but a machine taking random actions, no actual learning happening. The reward given at each step would not affect its future decision in any way. As epsilon approaches 0, the learning curve should be steep, with a high chance of using the best known path. The values approaching 1 would not approach the maximum angle available in the simulation conditions. This is due to the high probability of taking a random action which might decrease the energy of the system and therefore lowering the amplitude.

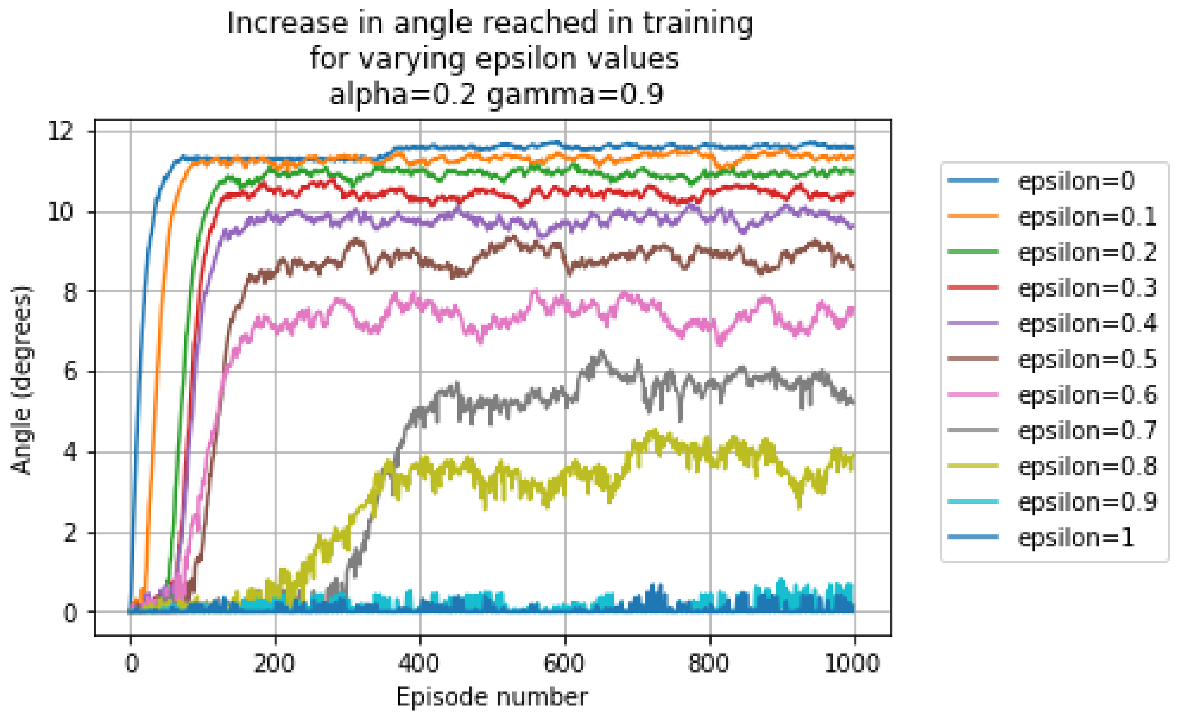


Figure 2.5: Angle achieved vs number of episodes for different values of epsilon

In Fig.2.5 it can be seen how the simulation respects the expected behaviour. In Fig.2.6 it is shown how the error for low epsilon values is big for the first episodes, when the system does not have yet enough data to decide what would be a good decision, but it goes towards 0 in the later episodes due to the learning curve. For the high epsilon, the error is consistent across the whole run. The initial error is smaller, but this can be expected from the already smaller value of the maximum angle, 3 degrees out of 3 giving a 100% variation while 5 degrees out of 12 giving a 41% variation.
For $epsilon = 0$ it is a matter of chance whether the system will learn at all. It encourages best-action decision all the time, there would be no exploration. This makes the case a fully-deterministic one, the agent taking the actions one by one,in order, until it finds something that works. This gives a high error, in some cases failing to reach high angles. It happens due to the fact that the lack of randomness gets it stuck with an action. With everything leveled to 0, the first value that gives a positive reward will be seen as the best one all the time for a given state which stops it from taking the real best decision. Sometimes that might be the best action, which gives a good training as it shown in Fig.2.5, while sometimes it might be the second best but it will stay stuck there.

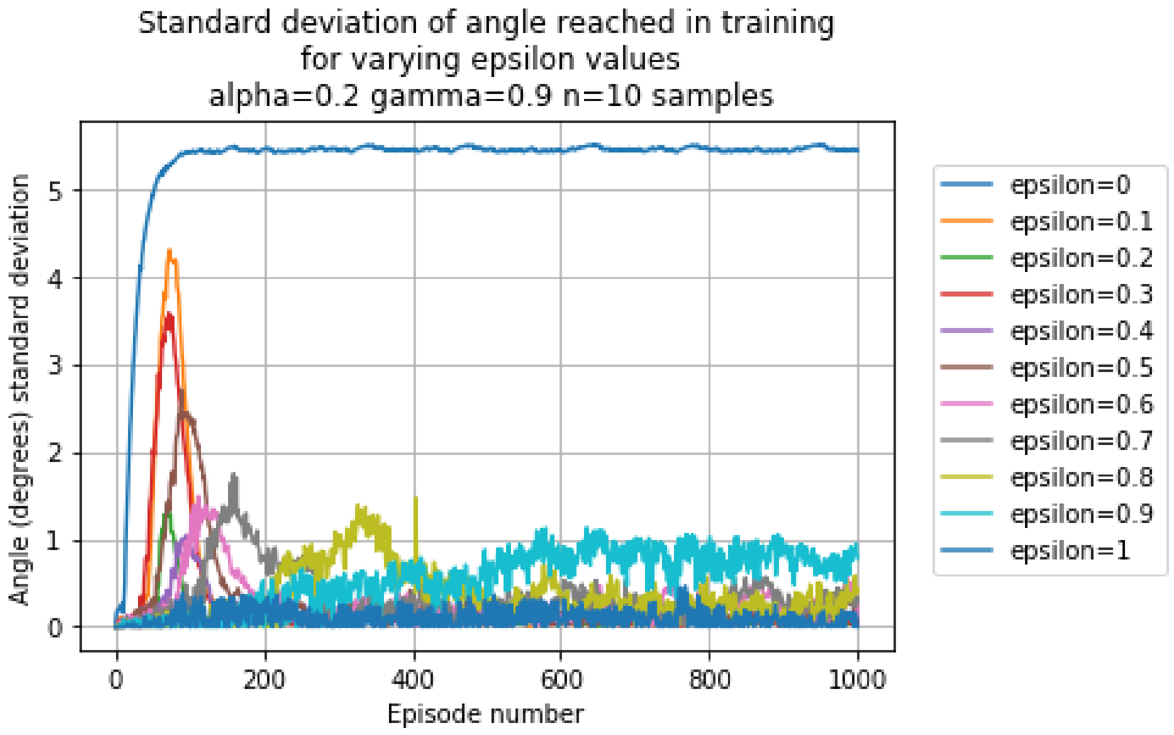


Figure 2.6: Angle achieved vs number of episodes for different values of epsilon

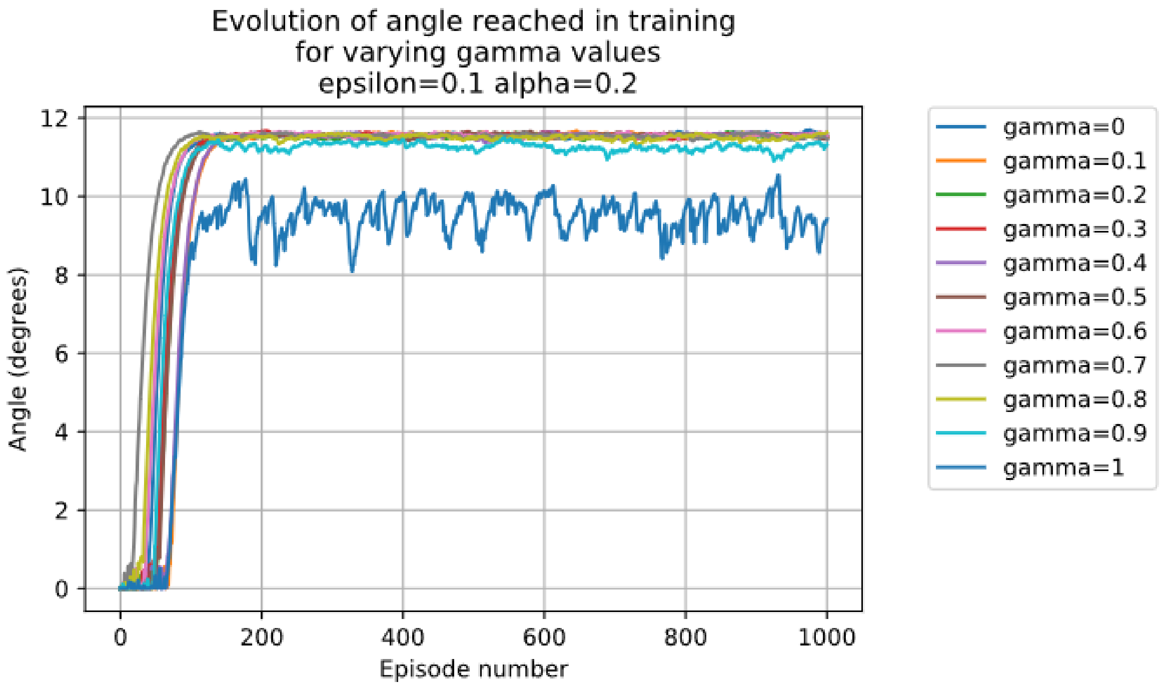


Figure 2.7: Angle achieved vs number of episodes for different values of gamma

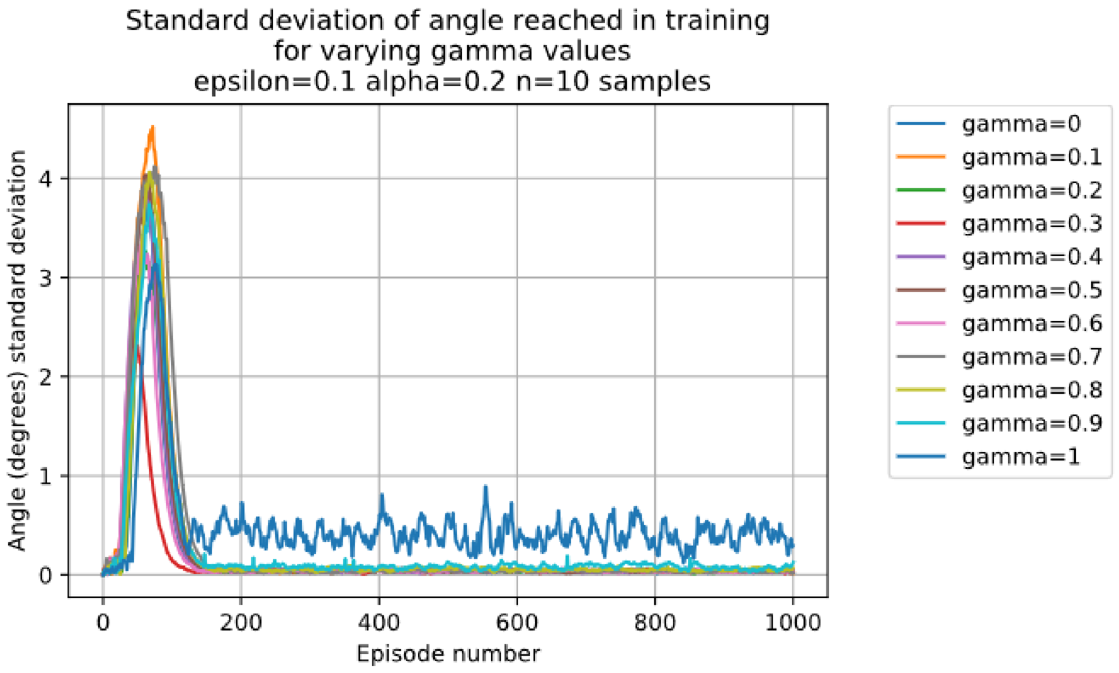


Figure 2.8: Standard deviation of angle achieved vs number of episodes for different values of alpha

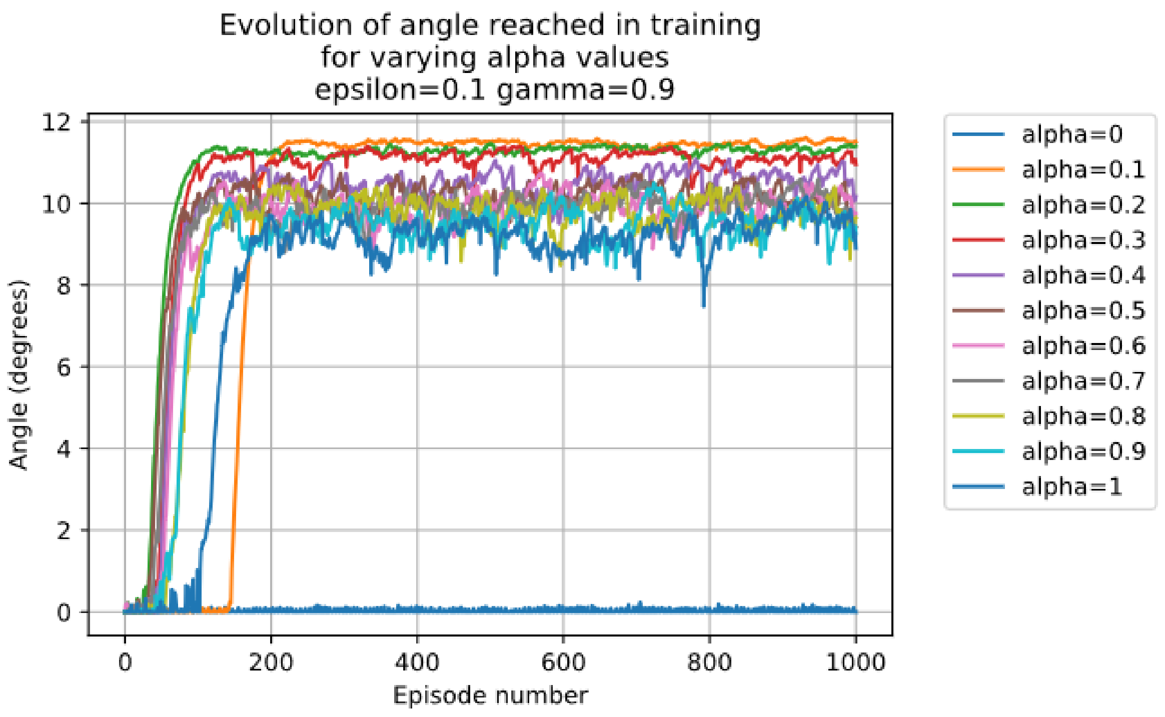


Figure 2.9: Angle achieved vs number of episodes for different values of alpha

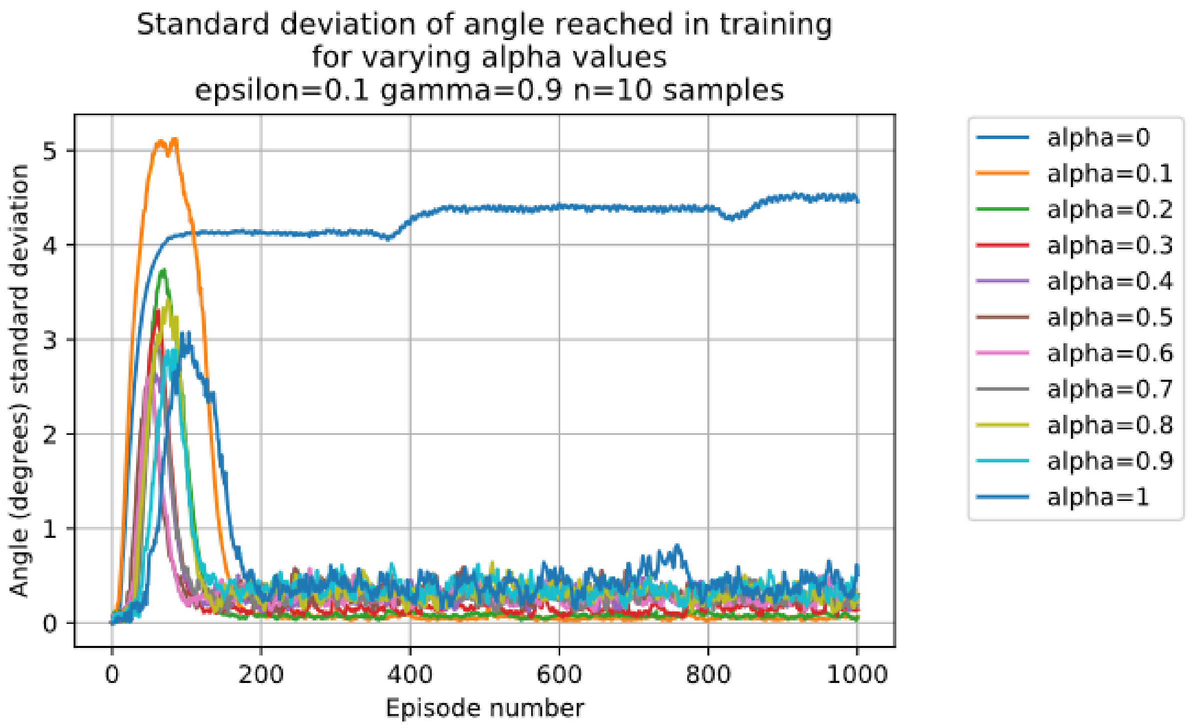


Figure 2.10: Standard deviation of angle achieved vs number of episodes for different values of alpha

Given that gamma values,discount factors, tell how important is the next maximum step, Fig.2.7 manifests predicted behaviour. A $\gamma = 1$ means the next choice would always be the immediate best, but that does not necessarily means the best in long term. This is similar to what was expected from the epsilon

policy.
The alpha value is expected to give better results as it approaches 0. A bigger value would decrease the training time, but would lower the maximum angle achieved. As it can be seen in Fig.2.10, an $\alpha = 0.1$ gives the highest angle, but it's the last one to achieve it, after almost 150 episodes.
The errors in Fig.2.10 fit the expected behaviour with high errors for low training rates in early episodes. A learning rate of 0 offers random behaviour so the angle error magnitude is at peak constantly. Going towards 1, the errors are smaller in the early stages, but do not as close to 0 as $\alpha = 0.1$.

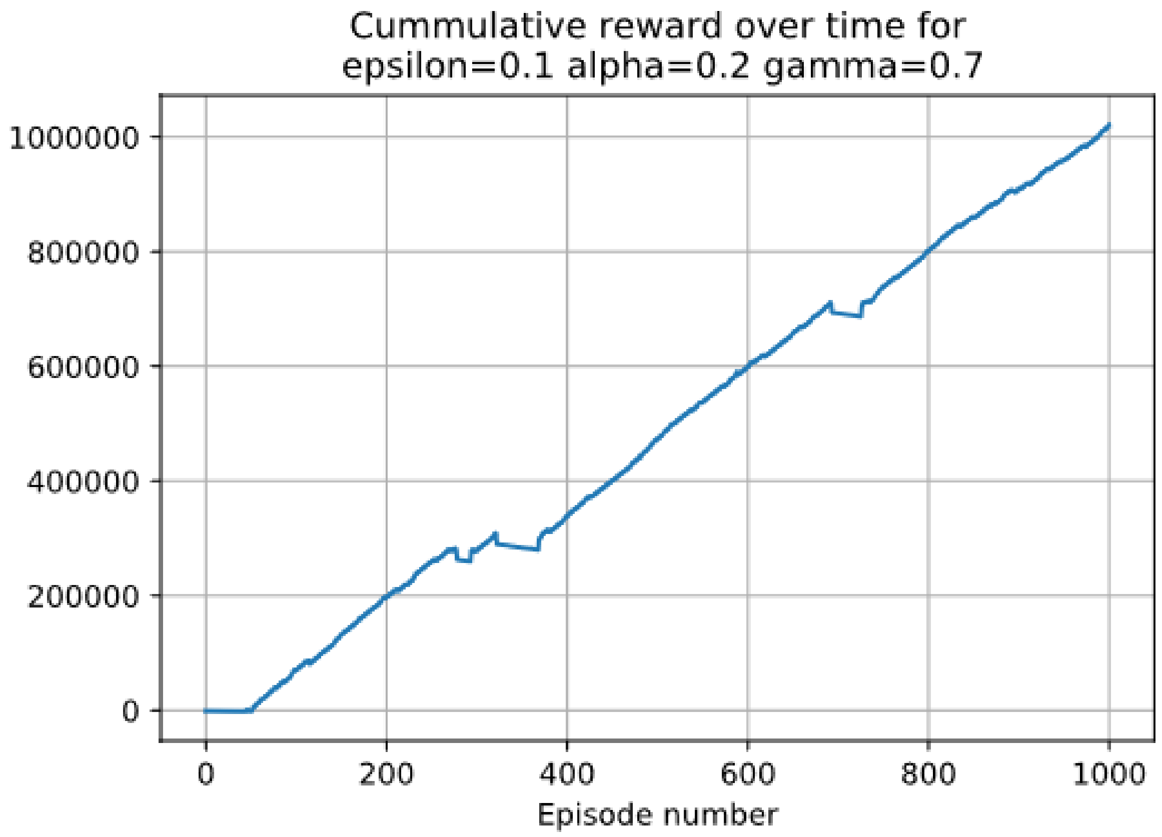


Figure 2.11: Cumulative reward as a function of number of episodes

The cumulative reward over time shown in Fig.2.11 shows linear behaviour. This is expected in a system that trains well, with three points where it gets stuck in a loop of bad decisions for a number of episodes. This is due to the choice of epsilon and the chance of choosing a random next action instead of the best one.

2.5.2 Implementing on NAO

The resulting Q-table from the simulation was successfully applied on the NAO robot.
For the rotational swinging, the maximum angle reached was close to the expected value obtained by the Code team with a deterministic algorithm. The expected value was 13 degrees, while the angle obtained with the Q-table is 10 degrees.
The difference is given by the choice of states, which is simplistic, combined with a delayed response time from the robot. This can be further improved by a better state-action table.
Parametric swinging does not self start,so a push is necessary, hence the higher starting angle.The angle reached in this case was far less than the expected value, which was supposed to be better than the

rotational one. On top of the mentioned reasons, simplistic model and delayed response, technical problem were encountered with the NAO robot. One of its legs would get stuck which would interfere with the swinging outcome.

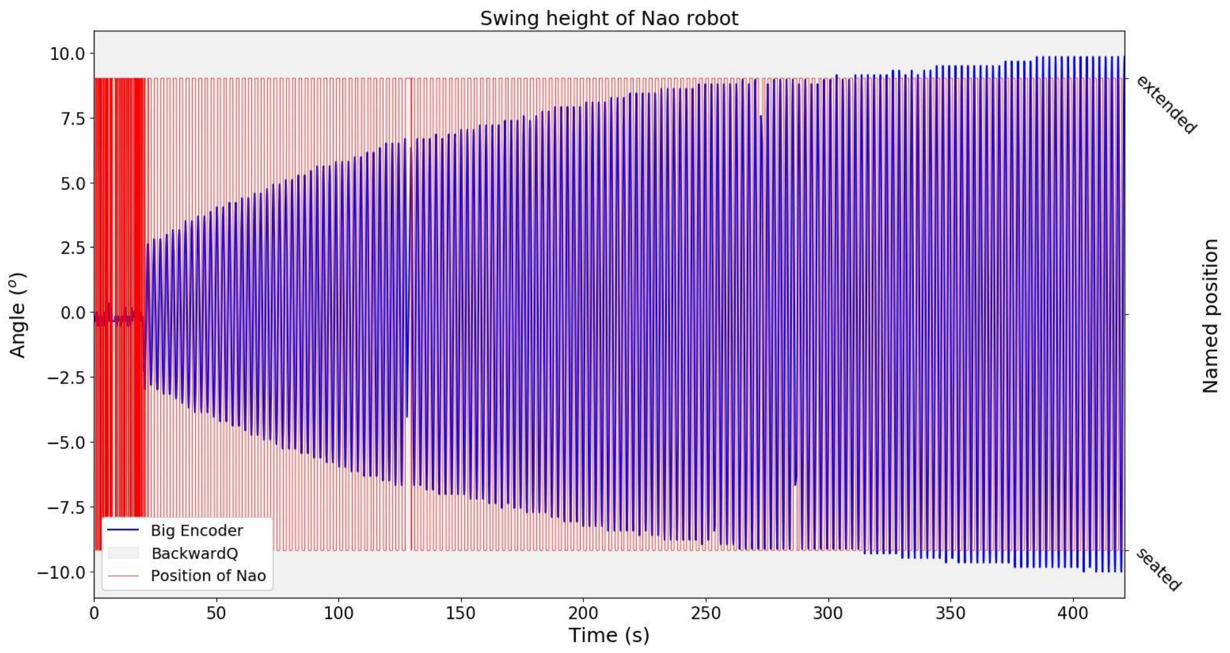


Figure 2.12: Angle achieved by applying rotational Q-table to the NAO robot

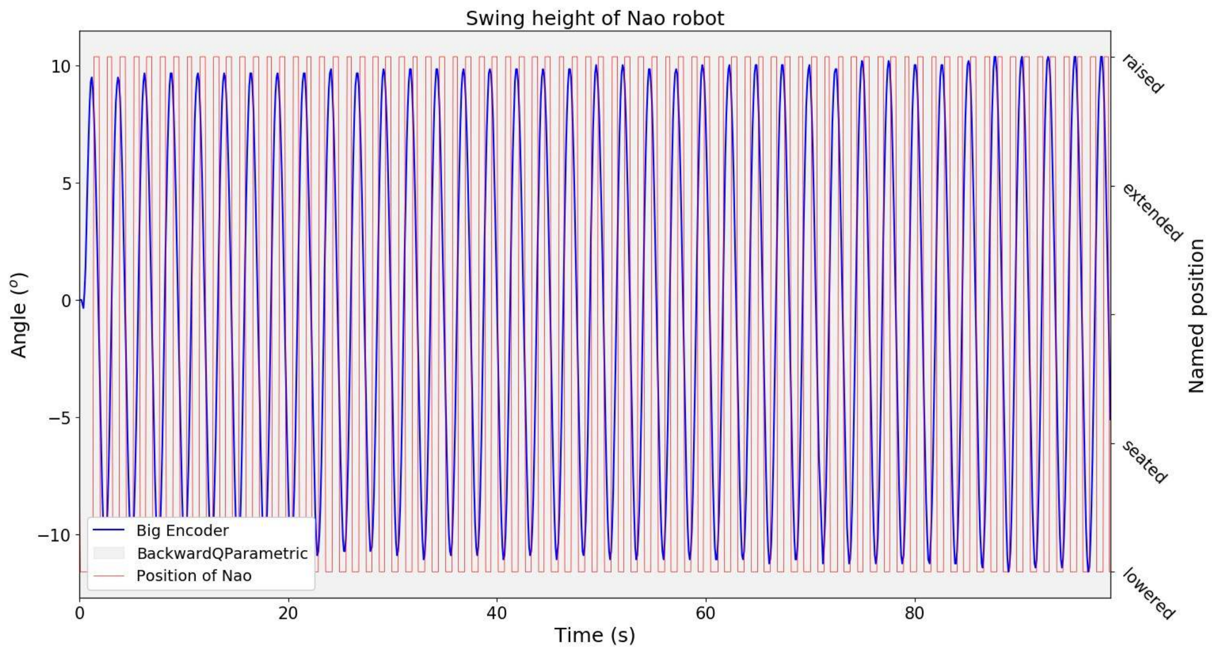


Figure 2.13: Angle achieved by applying parametric Q-table to the NAO robot

A model was tried in the simulation that might account for the delay response by having separate states for right before the movement is supposed to change. This would give the robot a chance to learn that attempting a movement before its velocity changes direction can give better rewards. A successful Q-table was not created with this model due to time constraints but it can be attempted in future years.

2.6 Conclusion

Fiesta Leung

2.6.1 Evaluation

The BQSA algorithm is a demonstration of the usefulness of integrating BQL with another RL algorithm. It converged to an optimal solution within a shorter period of time than the Q-Learning algorithm in both the parametric and seated swinging models of the Pymunk simulation. The optimal performance of BQSA is however yet to be tested.

The models of the Pymunk simulation were simplified to allow for easy simulation. It neglected the complex properties of the NAO robot as mentioned in section 16. The damping coefficient used was in the incorrect form and the centre of masses used did not reflect their exact positions in the swinging system in the laboratory. With such discrepancies, it was not surprising that the swinging motions of NAO carried out in the laboratory swinging system were not identical to those trained in the simulation. Essentially, the proxy environment that was developed to ease learning was optimised during training instead of the environment of interest (i.e. the swinging system in the laboratory). This was inevitable as the environment used in the simulation, even built as an exact replica, would have been overly complicated for the BQSA to run on. It would require the state space to be discrete in a considerably finer manner to match the complexity of the model of the environment. The reward received by the agent at each transition was defined by the change in the total energy of the swinging system associated with that transition. This was demonstrated to be more effective than defining the reward by the swinging angle the agent achieved in each transition, in guiding the agent to perform optimal pumping.

The discrete space applied was deliberately coarse to ease training, ignoring the continuous nature of the environment. Without a fine discrete system, state changes were not recreated as smoothly as in real life. Swinging motions performed by NAO were therefore not as smooth as desired. In addition to these omissions from the real-life model, the discount factors and the learning rates used in BQSA were not optimised due to time limitation. The corresponding optimal values for optimising the performance of BQSA in solving the swinging problem are yet to be determined. Furthermore, only one action-selection policy (i.e. ε-greedy policy) was investigated. The main drawback of the ε-greedy policy was its equal weighting for all actions during exploration. This implied the worst and next-to-best actions were chosen at an equal probability, ignoring their estimated values which signified the corresponding quality of the action [11]. It became undesirable in optimising the swinging motion as the worst actions would have a drastic effect in impeding the pumping of the swing, hindering the learning process.

In spite of all of these constraints, BQSA was good enough in producing a successful swinging model, generating results which agreed with those collected by the coding team within a reasonable margin of error.

2.6.2 Extension

The performance of BQSA could be improved in numerous ways. In terms of optimising the discount factors and learning rates, the most straight forward method would be determined through trial and error. Another modification would be to consider a dynamic reward function. Preferably, the reward signal should be updated and reinforced after each episode, using the information gained from the previous episode.

Two major extensions, made at the expense of the current simplicity, could be considered. First would be to devise an algorithm such that the agent learns to perform rotational and parametric pumping at small and large angles respectively. A more sophisticated discrete space would be required in order to guide the agent to execute the correct actions at the right angles. Alternatively, the motions of the upper and lower bodies could be defined separately, each with its own Q-table. This however, would lead to an unavoidable increase in the overall computational time due to the corresponding Q-value function being trained individually. The converged Q-tables could then potentially be merged after training to produce a better optimised swinging model. Second, would be to experiment on different action-selection policies. One potential candidate could be a softmax policy which utilises a temperature-dependent probability distribution (i.e. the Boltzmann distribution). A softmax policy addresses the problem of the ε-greedy policy mentioned above by ranking and weighting actions based on their estimated values [11].

With the aforementioned suggestions, the learning process could potentially be accelerated while also achieving a better final performance. Additionally, the SA-Q-Learning algorithm proposed by Guo et al. [13] has the potential to outperform the BQSA algorithm in optimising the swinging motion as was demonstrated in the cart-pole balancing problem in [10].

2.6.3 Summary

As an example of applying RL in robotics, a program featuring the BQSA algorithm was written in Python to optimise the swinging motions of the humanoid robot NAO. Simplifications were made in discretising the state space and the construction of the environment for simulation. The reward function was constructed based on the change in the total energy of the swinging system as it provided the agent a clearer guidance on achieving optimal swinging motions. The ε-greedy policy was chosen as the action-selection policy owing to its easy implementation. More importantly, it made sure there was an element of exploration to the agent's action-selection process.

Despite the inherent simplifications in the model, BQSA succeeded in training a swinging model which exhibited qualitatively similar pumping motions as those performed by NAO in the laboratory. BQSA was therefore concluded to be an effective learning algorithm in optimising the swinging motions of the NAO robot.

References

[1] M. Sebag, “A tour of Machine Learning: an AI perspective,” *AI Communications*, vol. 27, no. 1, pp. 11–23, 2014.

[2] Thomas, *RoboCup 2011 Robot Soccer World Cup XV*. Springer, 2012.

[3] H. Liu, N. Kubota, X. Zhu, R. Dillmann, and D. Zhou, *Intelligent Robotics and Applications*. Springer International Publishing, 2017.

[4] T. Dorazio and G. Cicirelli, “Q-learning: computation of optimal q-values for evaluating the learning level in robotic tasks,” *Journal of Experimental Theoretical Artificial Intelligence*, vol. 13, no. 3, pp. 241–270, 2001.

[5] S. Nagendra, N. Podila, R. Ugarakhod, and K. George, “Comparison of reinforcement learning algorithms applied to the cart pole problem,” *CoRR*, vol. abs/1810.01940, 2018.

[6] D. Butters, O. Diba, J. Forster, G. Hazar, Z. Hodgins, C. Hogg, E. Humphreys, H. Jacobs, P. Jones, M. Lim, and et al., “Robotics group studies teaching an aldebaran nao robot to swing,” 2016.

[7] K. Knee, H. Shaw, S. Bull, H. Gaskin, J. Morris, C. Patmore, H. Pratten, M. Hadfield, J. Torbett, D. Corless, and et al., “Optimising swinging motion with a nao robot,” Mar 2017.

[8] E. Tregoning, H. Birks, H. Bithray, M. Elliott, B. Adams, T. Batchelor, R. Bouab, R. Dubois, K. Giddha, J. Goldberg, and et al., “Optimising swinging motion of a nao aldebaran robot on a hinged swing,” Mar 2018.

[9] M. J. Mataric, *A comparative Analysis of Reinforcement Learning Methods*. 1991.

[10] Y. H. Wang, T.-H. S. Li, and C. J. Lin, “Backward q-learning: The combination of sarsa algorithm and q-learning,” *Engineering ApplicationsofArtificial Intelligence*, vol. 26, p. 21842193, Oct 2013.

[11] R. S. Sutton and A. G. Barto, *Reinforcement learning: an introduction*. The MIT Press., 2018.

[12] K. S. Hwang, S. W. Tan, and C. C. Chen, “Cooperative strategy based on adaptive q-learning for robot soccer systems,” *IEEE TRANSACTIONS ON FUZZY SYSTEMS*, vol. 12, Aug 2004.

[13] M. Guo, Y. Liu, and J. Malec, “A new q-learning algorithm based on the metropolis criterion,” *IEEE Transactions on Systems, Man and Cybernetics, Part B (Cybernetics)*, vol. 34, no. 5, p. 21402143, 2004.

[14] C. J. Watkins and P. Dayan, “Q-learning,” *Machine learning*, vol. 8, no. 3-4, pp. 279–292, 1992.

[15] G. A. Rummery and M. Niranjan, “On-line Q-learning using connectionist systems,” Tech. Rep. TR 166, Cambridge University Engineering Department, Cambridge, England, 1994.

[16] S. Zou, T. Xu, and Y. Liang, “Finite-sample analysis for SARSA and q-learning with linear function approximation,” *CoRR*, vol. abs/1902.02234, 2019.

[17] Z. Sadeghi, “Cliff walking problem,” 01 2009.

[18] R. E. Ambrose, B. E. Pfeiffer, and D. J. Foster, “Reverse replay of hippocampal place cells is uniquely modulated by changing reward,” *Neuron*, vol. 91, p. 11241136, Sep 2016.

[19] T. Haga and T. Fukai, “Recurrent network model for learning goal-directed sequences through reverse replay,” *eLife*, vol. 7, p. 131, 2018.

[20] K. Zhang and W. Pan, "The two facets of the exploration-exploitation dilemma," *2006 IEEE/WIC/ACM International Conference on Intelligent Agent Technology*, p. 17, 2006.

[21] S. Ravichandiran, *Hands-On Reinforcement Learning with Python.* Packt Punlishing, 2018.

[22] D. Apetroaei, "Robotics 2019." "`https://github.com/verlorenimmeer/Robotics2019`", 2019.