Deep Reinforcement Learning for a Four Degree of Freedom Robot Arm Control
Simulation accelerated by Human Demonstrations

by

Kirk Duran

A thesis submitted to
Sonoma State University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

in

Computer and Engineering Science

Committee Members:
Dr. Sudhir Shrestha, Chair
Dr. Mohamed Salem
Dr. Nansong Wu

June 5, 2021
Copyright 2021

By Kirk Duran
Authorization for Reproduction of Master’s Thesis

I grant permission for the print or digital reproduction of this thesis in its entirety, without further authorization from me, on the condition that the person or agency requesting reproduction absorb the cost and provide proper acknowledgment of authorship.

I grant permission for the print or digital reproduction of parts of this thesis without further authorization from me, on the condition that the person or agency requesting reproduction absorb the cost and provide proper acknowledgment of authorship.

Permission to reproduce this thesis in its entirety must be obtained from me.

Permission to reproduce parts of this thesis must be obtained from me. I do not approve the reproduction of this thesis, either in part or in its entirety.

Date: 

Name:
I dedicate this to every student and teacher that continues to pursue learning in this time of adversity. Also to my fiance Anna, who has given me infinite support during this research.
ABSTRACT

Purpose of the Study: The purpose of this study was to explore the efficacy of controlling a four degree of freedom (4-DOF) robot arm simulation with a deep reinforcement agent. Furthermore, I studied the effect that human demonstrations had on learning ability and execution performance.

Procedure: To determine the effect of reinforcement learning on controlling a 4-DOF robot arm simulation, I built a robot arm control library. I used a PhantomX Reactor Robot Arm. I defined Denavit-Hartenberg parameters for the robot arm, to calculate end effector Cartesian position via forward kinematics. I limited possible moves to incremented our joints by -1, 0, or +1. I built a dueling double deep Q value neural network to be trained by random exploration examples. I defined a simple reward function, +1 or -1, based on the distance between the end effector and the target position, and utilized the greedy reward policy. I trained our agent for 5000 episodes and recorded our max episode reward and in which episode it occurred. I then evaluated on target execution for various radius values. Next, using our robot arm I moved our robot arm by hand to generate a data set of human examples. I used these examples to retrain our agent and compared performance to our non-human trained agent.

Findings: When taking 300 steps for 5000 training episodes, our non-human trained learning agent showed a maximum reward peak of 300 at episode 4328. When taking 1000 steps for 5000 training episodes, our non-human trained learning agent showed a maximum reward peak of 450 at episode 4001. When taking 300 steps for 5000 training episodes, our human demonstration trained learning agent showed a maximum reward peak of 300 at episode 4156. When taking 1000 steps for 5000 training episodes, our human trained learning agent showed a maximum reward peak of 600 at episode 1537. When executing a move to random position task, our non-human trained robot agent reached on target position, with an error radius of 5 cm, 48.8% of the time. With an error radius of 1 cm, our non-human trained robot agent reached on target position, 11.2% of the time. For the same task, our human demonstration trained robot agent reached on target position, with an error radius of 5 cm, 92.7% of the time. With an error radius of 1 cm, our non-human trained robot agent reached on target position, 4.4% of the time.

Conclusions: Our results show that a reinforcement agent can effectively control a 4-DOF robot arm simulation. Our human trained agent showed greater reward performance at the beginning of training and a sooner occurring maximum reward value. However, our training rewards showed drop off which suggests our data set does not explore enough of our state action space. The human trained robot agent outperformed our non-human robot agent with a 5 cm radius, pointing to the strength of human demonstrative training.

MS Program: Computer and Engineering Science
Sonoma State University
Date:__________
ACKNOWLEDGMENTS

I would like to thank the members of my thesis committee. Professor Sudhir Shrestha, for his patience, flexibility, and continued support during these uncertain times, Professor Mohamed Salem, for his great suggestions when I was too close to the project to see new ideas, Professor Nansong Wu, for challenging me to dive deeper into background theory when I thought I knew enough. With their combined support, I was able to pour myself into this research.

A special thank you to the Intelligent Systems Lab for lending me the PhantomX Reactor robot arm, which served as the model for the research and coding achieved in this thesis.

I would like to thank Professor Farid Farahmand and the Engineering Science Department for creating and foresting an environment that promotes diverse research and collaboration.

I would like to acknowledge Dr. Shahrad Jamshidi and my colleague Olivia Piazza for pulling me into the field of machine learning and exposing me to new ideas which allowed me to grow as a researcher and scientist.

Finally, I would like to thank Sonoma State University.
# TABLE OF CONTENTS

Acknowledgments ................................... vi

List of Tables ...................................... x

List of Figures ..................................... xi

Chapter 1: Introduction and Background ................. 1

1.1 Motivation ................................... 1

1.2 Introduction .................................. 1

1.3 Literature Review ................................ 2

Chapter 2: Theoretical Analysis ......................... 5

2.1 Reinforcement Learning ............................ 5

2.1.1 Markov Decision Process ........................ 6

2.1.2 Q Learning .................................. 6

2.2 Robot Arms .................................... 8

2.2.1 Robot Arm Control ............................ 8

2.2.2 Denavit-Hartenberg Parameters .................. 8

2.3 Deep Reinforcement Learning ....................... 10

2.3.1 Neural Network ................................ 10
Chapter 3: Methodology ........................................... 16

3.1 Research Objectives ........................................... 16
3.2 Experimental Setup ........................................... 16
  3.2.1 Robot Arm ................................................. 16
  3.2.2 Control Scheme for Robot Arm ....................... 17
  3.2.3 Forward Kinematics ....................................... 17
  3.2.4 Robot Agent ................................................ 19
  3.2.5 Error Function ............................................. 19
  3.2.6 Replay Memory ............................................ 20
  3.2.7 Neural Network ............................................ 20
  3.2.8 Training .................................................... 21
  3.2.9 Human Demonstrations ................................. 21
  3.2.10 Human Retraining ....................................... 22
3.3 Study Design ................................................... 22
  3.3.1 Training Rewards ......................................... 22
  3.3.2 Move Accuracy ............................................ 22

Chapter 4: Results and Discussion .............................. 25

4.1 Training Rewards ............................................. 25
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.2</td>
<td>Execution Examples</td>
<td>27</td>
</tr>
<tr>
<td>4.3</td>
<td>Execution Accuracies</td>
<td>32</td>
</tr>
<tr>
<td>4.4</td>
<td>Comparisons to Prior Work</td>
<td>34</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Robot Arm</td>
<td>34</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Neural Network</td>
<td>35</td>
</tr>
<tr>
<td>4.4.3</td>
<td>Policy and Rewards</td>
<td>35</td>
</tr>
<tr>
<td>4.4.4</td>
<td>Human Involvement</td>
<td>35</td>
</tr>
<tr>
<td>5.1</td>
<td>Conclusion</td>
<td>36</td>
</tr>
<tr>
<td>5.2</td>
<td>Future Work</td>
<td>37</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Robot Arm</td>
<td>37</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Model Free Agent</td>
<td>37</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Reward and Policy</td>
<td>37</td>
</tr>
<tr>
<td>5.2.4</td>
<td>Data Set</td>
<td>37</td>
</tr>
<tr>
<td>5.2.5</td>
<td>Natural Language Processing</td>
<td>38</td>
</tr>
<tr>
<td>Appendix A</td>
<td>Experimental Equipment</td>
<td>40</td>
</tr>
<tr>
<td>Appendix B</td>
<td>Code</td>
<td>41</td>
</tr>
<tr>
<td>References</td>
<td></td>
<td>48</td>
</tr>
</tbody>
</table>
LIST OF TABLES

3.1 DH Parameters .............................................. 18
LIST OF FIGURES

2.1 Intelligent Agent ............................................. 5
2.2 Dueling Double Deep Q Network ................................. 7
2.3 Denavit-Hartenberg Convention .................................. 9
2.4 Neural Network ................................................. 10
2.5 Dueling Double Deep Q Network ................................ 12
2.6 Dueling Double Deep Q Network ................................ 13
2.7 Dueling Deep Q Network ........................................ 14
3.1 Robot Arm ..................................................... 17
3.2 Joints ......................................................... 18
3.3 Error Function ............................................... 20
3.4 Dueling Double Deep NN ...................................... 23
3.5 Human Recording Example 1 .................................... 24
4.1 4-DOF Robot Agent Training .................................. 25
4.2 4-DOF Human Agent Training .................................. 26
4.3 4DOF Robot Agent Training .................................... 26
4.4 4DOF Robot Agent Training Human ............................ 27
4.5 Robot Execution 1 .............................................. 28
List of Abbreviations

**DOF**: Degree of Freedom

**DH parameter**: Denavit-Hartenberg Parameter

**ELU**: Exponential Linear Unit

**R**: Revolute

**DQN**: Deep Q Neural Network

**RL**: Reinforcement Learning

**MDP**: Markov Decision Process
CHAPTER 1
INTRODUCTION AND BACKGROUND

1.1 Motivation

Robotics have been adopted by many companies to assist humans in applications that are tedious, labor intensive, or dangerous. However, for a given application, resources are spent on acquiring a robot and a robotic engineer to develop a solution for that application and configuration. The robot engineer is responsible for establishing the communications, scripting the controls, calculating the coordinate transformations, and programming the error handling. Then usually a technician is responsible for the daily operation or the robot operates independently. However, when processes change or requirements change, the resources already invested in acquiring and developing a robotic solution are not easily adapted to a new configuration or application without the help of a robotic engineer.

Rather than having a robotic engineer hard code the operation of a robot for a new application, companies could utilize deep reinforcement learning to train an intelligent agent to operate a robot for that specific application. This would allow resources companies invest in robotics to be more robust for different applications and purposes.

Furthermore, there are several applications where humans are still the best at executing the task, like surgery, feeding, pathing, and alignment. However, by building a data set of human experience I could train an intelligent agent based on these examples to create a more human-like robot.

1.2 Introduction

This thesis aims to explore the ability of an intelligent agent to control a 4-DOF robot arm to reach a given target position. More specifically, this thesis aims to characterize the
efficacy of deep Q Learning in a very large state-action space where simple q learning is prohibited. Furthermore, I will explore the effect that a human generated data set has on an intelligent agent’s learning ability.

The remainder of this thesis is divided into five sections, as follows:

Chapter 2 explains the background theory for reinforcement learning, how reinforcement learning is formalized and how it is not appropriate for our experiment. Chapter 2 also explains robot configuration and the difficulty of controlling high degrees of freedom robot arms. Chapter 3 provides an explanation of experimental setup, the decisions I made on our set up, how our process works, and the metrics used to evaluate performance. Chapter 4 contains the results for both the non-human trained agent and human trained agent and discussion. Chapter 4 also compares previous results with the results obtained using the methods described in this paper, highlighting the differences in outcomes as well as any limitations of current methods. Chapter 5 presents conclusions drawn from the results and give an insight to the future work.

1.3 Literature Review

Recently researchers have focused on autonomous robotics, with emphasis being placed on task specific training. It has been shown that I can remove the complexity of workspace calculations and limitations of configuration by solely relying on external images or raw positional sensor output. In [1] a simulation of a three joint robot arm was trained to reach a target position in two dimensions. The feedback used was external visual observation of the end effector. This allowed [1] to develop a model free approach to their experiment. They successfully trained a Deep Q Network to control their robot. However, once leaving simulation for a real robot arm, the algorithm failed to perform. Also, due to the complexity of higher dimensional application, dimensional constraints limited the robot arm to two dimensions. In [2] a simulated robot arm was trained using deep reinforcement learning to locate and grasp a cube in one dimension. The loss function is calculated using images of
the environment to determine the difference between the end effector and cube. They use a convolutional neural network to identify the end effector position. The agent was awarded immediately after each move. They showed the robot agent was able to steadily improve Q values showing a true improvement. While the study was limited in scope they went on to explain expanding it to higher dimensions was possible. In [3] the researchers pointed out the difficulty in tuning robotics for precision manufacturing assembly. They focused on deep reinforcement learning to insert a peg into a hole task. The robot was holding the peg and performing a search to find a lateral position that would fall into the holes clearance. Then the agent performed the insertion phase. They used a 7-DOF robot where the joint sensor positions was the feedback. They also used a recurrent neural network in their reinforcement learning. They showed the robot successfully completed the insertion task with little angular errors. However, the majority of the task occurred in one dimension. In [4] the research focused on a grasping task. Seven robot arms where trained using computer vision and Q-learning in one dimension to grasp various objects. They trained on 580,000 real grasp attempts and saw a 96% success rate across a wide range of unseen objects.

Also, in an effort to produce robot agents that are robust to variations, some research incorporated perturbations into their models. In [5] the focus was using deep reinforcement to learn trajectory execution on two different simulated robot arms with different architectures. They used a 3-DOF robot arm and used images of the end effector for feedback. They also limited the number of state-action spaces by limiting changes of the joint angles to discrete +/- 5 degree moves. They showed that both differing robot arms performed well with the same hyper parameters. The researchers hoped the agent could overcome perturbations like increasing link length. However, they saw the agent’s error increased with link length. In [6] the research was focused on real world training of robot arms to open a door. To speed up the training multiple 7-DOF robots were trained simultaneously and pool their updates asynchronously. It took two workers 2.5 hours to reach 100% accuracy learning on a Normalized Advantage Function using the joint and end effector positions as state
representation. In [7] model free deep reinforcement learning was used to train a robot on multiple policies like insertion and collision avoidance. Soft Q learning was used to allow for learning a single policy. Then the researchers were able to combine policies together and saw their agent place the object correctly and avoid the collision. They showed that soft Q-learning outperformed other forms of learning. They showed the value of composing policies for different tasks but also pointed out further corrections were needed to reduce bias in the composed Q-function which would allow any policies to be combined.

Additionally, there have been studies where human demonstration has been incorporated into the experiment. In [8] a simulated robot arm was tasked with generating trajectories. A non-expert human was used for feedback by defining goals; comparing and selecting amongst possible trajectories. The researchers showed the preference elicitation in reinforcement learning were successful in learning unknown reward functions. In [9] using a 7-DOF robot arm where a human physically moves a robot with their hands, feedback are joint position states, forces, and raw images. Learning occurred using a Deep Deterministic Policy Gradient with a sparse reward function. The task was inserting a clip in one dimension, so the problem was constrained in scope. However, the researchers showed demonstrations were a viable alternative to shaping rewards. In [10] they defined a modular set of policies based on human-like movement called “human priors.” The policies focused on total duration of the movement (Duration), intensity of the control put into the actuator (Energy), trajectory directness between starting point and target (Directness), smoothness of the end effector position (Smoothness), and smoothness of the joint positions (Continuity). The robot arm was simulated and it only focused on executing trajectories. They used a 3-DOF robot arm and trained a deep reinforcement learning agent that used the joint positions, angular velocity, and effector as the state positions. The performed a reaching task limit to two dimensions and identified that by focusing on some metrics, others were also tuned, showing an interdependence.
CHAPTER 2
THEORETICAL ANALYSIS

2.1 Reinforcement Learning

Reinforcement learning is a kind of machine learning, where an agent learns by interacting with an environment by executing an action, and based upon the change of state, a reward or punishment is given back to the agent (Figure 2.1). An agent is a program that can be trained to make intelligent decisions by observing the state of the environment, and selecting an action that maximizes desired reward uptake. The agent is the learner in reinforcement learning. An environment is what the agent can interact with. Our robot arm is our agent's environment. States are the positions in our environment that our agent can reach. The different permutations of joint positions are our states. The agent reaches different states by performing actions. Based upon the state reached and action performed the agent receives a numerical reward or punishment, to determine if that action is worth performing in the future. Reinforcement learning becomes a process of trial and error, where the agent randomly explores the state action space with a goal of maximizing rewards. How I define our agent reward seeking behavior in an environment is called a policy. For our agent I am implementing the greedy policy which always selects the action with the highest reward available [11].

Figure 2.1: Agent interacting with environment
2.1.1 Markov Decision Process

The Markov Decision Process (MDP) is a mathematical frame for long term decision making in uncertain domains where the outcomes are influence by an agent’s actions. An MDP framework contains the following [12]:

- State $s \in S$; set of all states
- Action $a \in A$; set of all actions
- Probability $p \in P$; set of transition probabilities from states $s$ to $s’$ by action $a$
- Reward $r \in R$; set of rewards for transition from $s$ to $s’$ by action $a$
- Policy $\pi(s) \rightarrow a$ is function that maps the state space to the action space

In MDP our goal is to find an optimal policy $\pi$. One example is a policy that maximizes our reward return, which for each state I need to find the action that maximizes the expected reward $E[r_t|\pi, s_t]$

In reinforcement learning the agent can learn [13] :

- A policy $\pi(s) \rightarrow a$
- A value function that gives us the expected return value $E$
- All probabilities $P$ for states $s$ to $s’$ by action $a$

2.1.2 Q Learning

To solve a MDP model I can apply a Q learning algorithm. In Q learning, there is an exploration phase, where our agent randomly explores the state action space. For a state $s$, a random action $a$, results in a transition to state $s’$, and produces reward $r$. 
Using the reward feedback, the agent assigns values to each state action pair determined by Equation 2.1.

\[ Q(s, a) = (1 - \alpha)Q(s, a) + \alpha[r + \gamma \max_{a'}Q(s', a')] \]  

(2.1)

These are called Q values, which inform the agent to the expected reward of taking an action from a given state. These values are stored in a Q value table (Figure 2.2). Q learning requires that an agent exhaustively explore the state action space to assign and update q values to every possible state action [14]. This training will generate a Q function that can be exploited in the exploitation phase, \( Q : S \times A \rightarrow R \).

![Figure 2.2: Q value table filled by our state-action exploration](image)

However, in a large state action space like a high degrees of freedom robot, it is inefficient to explore a state action space. The size of the Q value table is \( N \times M \), where \( N \) is size of set \( S \) and \( M \) is the set of set \( A \). For a large state action space, with a large Q value table, too much time is needed to search the table and too much memory is needed to store the table [15].
2.2 Robot Arms

2.2.1 Robot Arm Control

Robot arms have joints where the position or orientation of their stiff links can be controlled. Each of these joints is called an axis. If a robot arm has n axes, I could call it an n-DOF robot arm. Each joint can be rotational, called revolute (R), or linear, called prismatic (P). I can create an articulate coordinate robot arm by combining multiple R axes [16].

For a high degree of freedom robot arm, where each axis is R, when the angle $\theta_n$ of each joint is changed the Cartesian position of the end effector also changes. Our intelligent agent can control the robot arm via adjusting these joint angles. To reward or punish the agent, I need to know the distance between the end effector and the target position. One method would be to use external sensors or cameras, to measure the end effector position. However, given the joint angles I can apply forward kinematics to calculate the end effector position. For these calculations I can use Denavit-Hartenberg Convention, which removes ambiguity from our positions.

2.2.2 Denavit-Hartenberg Parameters

The Denavit-Hartenberg (DH) convention defines a configuration for our robot arm. Between two axes we are able to define four parameters $\alpha_i, \theta_i, a_i, d_i$ (Figure 2.3), which describe the link twist, joint angle, link length and link offset [17]. For each joint, I have a homogeneous transformation matrix $A_i$, Equation 2.2, which utilizes our DH parameters to calculate the coordinates and orientation of our next joint or end effector.
Figure 2.3: Denavit-Hartenberg convention between two joints

\[
A_i = \begin{bmatrix}
\cos(\theta_i) & -\sin(\theta_i)\cos(\alpha_i) & \sin(\theta_i)\sin(\alpha_i) & a_i\cos(\theta_i) \\
\sin(\theta_i) & \cos(\theta_i)\cos(\alpha_i) & -\cos(\theta_i)\sin(\alpha_i) & a_i\sin(\theta_i) \\
0 & \sin(\alpha_i) & \cos(\alpha_i) & d_i \\
0 & 0 & 0 & 1
\end{bmatrix}
\] (2.2)

I can then use the matrix multiplication of our homogeneous transformation matrices for each joint to calculate the end effector position from our transformation matrix, \( T_j \) Equation 2.3.

\[
T_j = A_{i+1}...A_j = \begin{bmatrix} R_{i}^{i-1} & O_{i}^{i-1} \\ 0 & 1 \end{bmatrix}
\] (2.3)

where

\[
O_{i}^{i-1} = \begin{bmatrix} x_{\text{tip}} \\ y_{\text{tip}} \\ z_{\text{tip}} \end{bmatrix}
\] (2.4)
2.3 Deep Reinforcement Learning

Deep reinforcement learning solves the difficulty of our large state action space by combining deep learning and reinforcement learning. Instead of exploring our entire state action space I can randomly explore just a small subset of our state action space and record those as examples in our replay memory. I then sample those memories to train a neural network that will predict Q values for a given state [14].

2.3.1 Neural Network

A neural network is an approximation function. If a function $f(x) = y$ exists but is unknown or hard to calculate, I can train a neural network to act as an approximation function $f^*(x, \theta) = y$. Where $\theta$ are the weights and bases of our neural network. I can train this neural network using known input and output pairs. As our neural network guesses I can compare our predictions against the known true values to generate a loss value, which will determine how our optimizer adjusts our parameters until our approximation function meets some satisfying accuracy (Figure 2.4) [18].

![Figure 2.4: Neural Network](image)
### 2.3.2 Deep Q Learning

For our environment’s Q values I can create a DNN that finds an approximation function $Q_\theta$, called a Deep Q Network (DQN). Utilizing a DQN to predict our Q values is called Deep Q Learning.[19] I train our DQN by minimizing the loss between our predicted Q value and true Q value in equation Equation 2.7 [20].

\[
L(\theta) = Q^*(s, a) - Q_\theta(s, a)
\]  

(2.5)

\[
L(\theta) = r + \gamma \max_{a'} Q(s', a') - Q_\theta(s, a)
\]  

(2.6)

\[
L(\theta) = r + \gamma \max_{a'} Q_\theta(s', a') - Q_\theta(s, a)
\]  

(2.7)

In deep Q Learning our agent only explores a fraction of our state-action space and stores those explorations in replay memory as examples to train a Q value neural network [20]. However, as I explore our environment I also want to train our Q value predictor neural network. I can utilize the Epsilon-Greedy Algorithm, where an epsilon value, $\epsilon$, defines the probability whether the agent’s next phase will be an exploration or exploitation phase [11]. If our agent enters an exploration step, our next action will be random. If our agent enters an exploit step, our Q value neural network will predict the Q values for each action. Then our greedy policy will choose the action with the highest predicted Q value. I can also use a decaying $\epsilon$ value to emphasize exploration at the beginning of training and exploitation towards the end [11].

After each learning episode, if there is a new max reward, the Q value networks weights are set to the active weights from when the new max reward was reached. Then new batch training of our neural network is executed (Figure 2.5).
2.3.3 Double Deep Q Learning

With DQL, there can be faults caused by overestimation. The Target Q value is given by:

\[ y = r + \gamma \max_{a'} Q_\theta(s', a') \]  
(2.8)

Since I force maximum Q value to be selected by \( \max_{a'} Q_\theta(s', a') \), the agent can learn this action as a preference, where the Q value can be too high [14].

I can solve this by implementing Double Deep Q Learning, where I create a duplicate of our neural network. One neural network predicts the action and the second predicts the Q value (Figure 2.6).

One neural network is parameterized by our network parameter \( \theta \), and predicts the action that has the maximum q value Equation 2.9.

\[ y = r + \gamma Q'_\theta(s', \max_{a'} Q_\theta(s', a')) \]  
(2.9)

The second neural network is parameterized by \( \theta' \) predicts the Q value for that action.
Figure 2.6: Double Deep Q value prediction neural network, with an action network and a q value network

Equation 2.10. [20]

\[ y = r + \gamma Q_{\theta}(s', a') \]  

The Q value neural network is updated much less frequently than our action neural network, but it is still a periodic copy. [11] The training implementation of the double deep Q learning agent is the same as regular deep q learning.

2.3.4 Dueling Deep Q Learning

In our state action space there could be several state action spaces I do not value at all, therefore the choice of action doesn’t matter. For instance a robot arm position an optic in a beam path should block the beam with one of its links. Also, in a large state action space there are state where the Q values for every action are equal [21]. I can use a dueling deep network to determine which states are not valued or which actions have value, by rewriting our q value as the sum of state values and action advantages as:

\[ Q(s, a) = V(s) + A(s, a) \]  

(2.11)
From the value function I can learn if a state has value without predicting the Q values for each action from that state. From the advantage function, I can determine if an action stands out or gives the same value as every other action [20].

![Dueling Deep Q value prediction neural network](image)

**Figure 2.7**: Dueling Deep Q value prediction neural network, fully connected layers for state value and action advantage

To build a dueling deep neural network that can output Q values like in Equation 2.12, I can construct a neural network with two outputs (Figure 2.7). Each output is a fully connected layer. The first output is a scalar output $V(s, \theta)$, The second layer outputs vector $A(s, \theta)$ [21]. I then aggregate the to arrive at Q values:

$$Q(s, a, \theta) = V(s, \theta) + A(s, a, \theta)$$  \hspace{1cm} (2.12)

However, this equation leads to unidentifiability, where different combinations of V values and A values give the same Q value. To solve this problem, I can constrain the advantage values to output zero, for the chosen action, $A(s, a) = 0$ [22]. Then the optimal policy will produce zero action advantage, such that the value of the state is equal to the q value as seen in Equation 2.13. Training our dueling deep neural network is the same as
for the DQN.

\[
Q(s, a, \theta) = V(s, \theta) + (A(s, a, \theta) - \max_{a'} A(s, a', \theta))
\]  
(2.13)

2.3.5 Deep Q learning from Demonstrations

Deep Q learning from Demonstrations (DQfD) is a form of imitation learning, where I want to agent to duplicate the behavior of an expert. Instead of learning from zero, I generate a data set of expert examples of the desired task. I store these examples in our replay memory, and pre-train the agent. During pre-training our DQN is trained by the expert examples. After pre-training, the agent begins the training phase, where the agent starts to generate its own examples. I continue to batch sample both the new random examples and expert demonstrations to update the neural network. I can also give more weight or access to our expert demonstrations, to create a prioritized replay buffer which will hopefully increase the imitation [20].
CHAPTER 3
METHODOLOGY

3.1 Research Objectives

- Objective 1: To create a control scheme for an existing 4-DOF Robot Arm in python utilizing a microcontroller and the PhantomX Reactor Robot Arm.

- Objective 2: To create a self-trained autonomous 4-DOF robot arm agent that will reach a given target position. I will achieve this by implementing a Deep Q Learning Algorithm, where I create a Deep Q value network to predict Q values for our agent. I will determine the best parameters for the neural network, and structure that maximizes learning.

- Objective 3: To create a human-trained autonomous 4-DOF robot arm agent that will reach a given target position. I will achieve this by creating a human demonstration data set and implementing a Deep Q Learning from Demonstration Algorithm, where I create a Deep Q value network to predict Q values for our agent. I will determine the best parameters for the neural network, and structure that maximizes learning.

3.2 Experimental Setup

3.2.1 Robot Arm

Our Robot Arm is a PhantomX Reactor Robot Arm [23]. Each joint is a revolute joint creating an R,R,R,R - waist, shoulder, elbow, and wrist configuration as seen in (Figure 3.1). In polar coordinates, our waist axis controls our $\theta$ position and our remaining 3 axes control our radius r and height h. The limits of each axis are: $\theta_1 = 300^\circ$, $\theta_2 = 180^\circ$, $\theta_3 = 200^\circ$, $\theta_4 = 210^\circ$ [23].
3.2.2 Control Scheme for Robot Arm

Our robot control script interfaces with a microcontroller to control the robot arm. Each joint is actuated by Dynamixel AX-12 servo motors, with 1024 bit position sensors, which results in approximately 0.3° position resolution. I wrote functions to convert between bits and degrees, when reading to and writing from the servo. The main functions utilized are move position, read position, disable torque, and enable torque.

To constrain our state action space I limited our possible actions to discrete -1,0, or +1 actions for each joint. With 4 joints and 3 possibilities, I have $3^4 = 81$ possible moves represented by a 4 digit ternary number or “trit”, where 0 is equal current joint position, 1 is +1 to current joint position, and 2 is -1 to current joint position. For example, 2102 results in $\theta_1 = -1^\circ$, $\theta_2 = +1^\circ$, $\theta_3 = +0^\circ$, and $\theta_4 = -1^\circ$.

3.2.3 Forward Kinematics

To utilize forward kinematics I defined the DH parameters shown in Table 3.1. To find our DH parameters I began by deciding orientations of our axes as in (Figure 3.1). Axis 1 is revolute around Z. Axis 2 has an angle twist $\alpha_2$ of 90° and the rest of the axes are parallel to Axis 2. Next I measured the link lengths and any axis offsets [24]. Due to the difficulty
of precise measurement, some of our link lengths are not exact, where small adjustments lead to larger deviations in our position calculation. For example, changing our 2nd link length from 14.5 cm to 14.6 cm changed our end effector position for $(\theta_1 = 39^\circ, \theta_2 = 110^\circ, \theta_3 = 50^\circ, \theta_4 = 75^\circ)$ by 0.3 cm.

With our DH parameters I am able to utilize forward kinematics to find our Cartesian coordinates of our end effector by applying Equation 3.1, which I calculated via matrix multiplication of each homogeneous matrix $A_i$ [17].

$$T_4^0 = A_1A_2A_3A_4 = \begin{pmatrix} R_4^0 & O_4^0 \\ 0 & 1 \end{pmatrix}$$

For example a position of $\theta_1 = 0^\circ, \theta_2 = 180^\circ, \theta_3 = 175^\circ, \theta_4 = 105^\circ$ gave us an end effector position of (45.2 cm, 0 cm, 6.8 cm) as I confirmed on our real robot arm.
3.2.4 Robot Agent

Our Robot Agent selects an action by observing the current state and evaluating available Q-values against the Greedy Reward Policy. For our agent, the Q values are not stored in a table but are predicted by our Deep Q Value Network based on the current state input.[22]

Our reward function is the distance from our end effector to our target position. If I are closer than the previous step, I reward the agent with +1. If I are farther away, I punish the agent with -1. I quit this episode and start the next episode if I meet any one of the end of episode conditions:

- If I reach a total reward of -50
- If I reach a total reward of 300
- If I are punished 10 times in a row
- If I are within some epsilon radius of the target position

A negative reward limit of -50 shows our agent isn’t taking the right path and is better off restarting. The limit of 10 bad moves in a row exists for a similar reason. A positive reward limit of 300 is due to the longest limit being $300^\circ$, which in the ideal case can be traversed in $300 \ 1^\circ$ increments. And finally, if I reach the target, I want to stop.

3.2.5 Error Function

For our error function, first I utilized euclidean distance between our end effector position and target position. However, I found when our target is 180 degrees away from our position, that a move in either direction rewards the agent equally as seen in (Figure 3.3). So, I altered our error function to be the shortest arc length plus the euclidean between the radius and height variables (Equation 3.2).

$$error = r \times \text{abs}(\Delta \theta) + \sqrt{\Delta r^2 + \Delta h^2}$$  (3.2)
3.2.6 Replay Memory

Our replay memory module is used to store example moves whether generated by random exploration or human demonstration. I made sure that the replay memory was large enough so that memories are not overwritten. The format of our replay memory is [starting state, action, reward, ending state, if done Boolean].

3.2.7 Neural Network

Our Q Value Network is a Dueling Double Deep Neural Network as seen in (Figure 3.4). Our input layer size is 7 for the state observation which is the x, y, z of our target position and $\theta_1$, $\theta_2$, $\theta_3$, $\theta_4$ of our robot arm. Our Neural Network outputs one state value and 81 Q value predictions; one for each of our actions. I have 5 hidden layers with 128 nodes each. I wanted a hidden layer node size greater than our output size of 81, so our network is not too small to learn. Usually, sizes are selected as powers of 2 [22]. I am using Exponential Linear Unit activation functions, since they do not suffer from vanishing gradients, exploding gradients, and have lower training times [18]. I chose the adam optimizer, due to integration in tensorflow and reliability in past deep reinforcement learning applications [25]. I chose the huber loss function due to its insensitivity to outlier data [26]. I used a learning rate of 0.001 and batch size of 64, found by trial and error.
3.2.8 Training

To train our neural network our agent self-generated a data set by executing random exploration moves. Our agent randomly selects 1 out of 81 possible actions. They check to see if the move would violate limit positions before execution. If so they select a different move. After execution, I calculate the end effector position and calculate the distance error from our target position. I then reward or punish our agent. Next I check if any end conditions have been met. Finally I store this information in our replay memory. During a training step I batch sample our replay memory to train our neural network. I input our examples into our neural network and compare the output predictions to our replay memory values and update our network parameters by our learning rate. I trained 5000 episodes, such that our agent was able to explore enough of our environment. Each episode executed 300 steps, the worst case ideal being 300 perfect action selections in a row. I also performed a second training with 1000 steps, such that our agent could explore more of our environment before being limited by end of episode conditions.

3.2.9 Human Demonstrations

To create a human demonstration data set I utilized our physical robot arm. I centered our robot arm on a grid with the theta 1 = 0 as the x axis. I established communications with the robot arm, enabled torque and moved to a randomly generated starting position for each joint. I generated a random target position and placed a marker at the physical coordinates. I then manually braced the robot arm and disabled torque. I then slowly guided the robot arm to the target position. As I physically guided the robot arm, I sampled and recorded the sensor position every 100ms (Figure 3.5). Again based on our longest limit, 300°, if I gently and slowly moved our robot arm over 30 seconds, I saw a 1° joint position change every 0.1 seconds. I only completed 100 example recordings, simply due to time and effort limitations. I processed our data set to match the formatting of our replay memory. I removed duplicates of states. I compared starting and ending positions of each step to
determine what the equivalent action was for that step. Since I assume a human would take
the most direct path, for every action in our recording I assigned a reward of +1.

3.2.10 Human Retraining

Once I have our human demonstration examples I retrain our neural network. However,
now instead of purely exploring our state-action space I utilize only our human demonstra-
tion data set, or a combination of human examples and exploration examples.

3.3 Study Design

3.3.1 Training Rewards

To evaluate the learning ability of our robot agent I recorded the total reward for each
episode of training. This allowed us to easily identify what our maximum total reward was
and how soon it occurred. For our neural network, the best weights were determined by the
maximum total reward episode weights. I compared the training performance between the
non-human trained agent and human trained agent.

3.3.2 Move Accuracy

Once I have a trained agent I can randomly generate target positions and starting positions
and allow the agent to attempt to reach the target. I defined an epsilon radius value to
determine if the agent reached an on target position. I had our agent execute 100 episodes
of move to random position and count how many executions reached an on target status. I
executed this test with 1 cm radius and 5 cm radius. I repeated this process with the human
example trained agent and compare.
Figure 3.4: Dueling Double Deep Neural Network
Figure 3.5: Human Demonstration Recording Example 1
CHAPTER 4
RESULTS AND DISCUSSION

4.1 Training Rewards

I trained our non human agent for 5000 episodes, with each episode having 300 steps (Figure 4.1). After each step I rewarded or punished our agent with -50 being the minimum and 300 being the maximum. I observed that our agent learns slowly with the maximum peak of 300 occurring on episode 4328.

![Rewards for 4D Training with No Human Experience](image)

Figure 4.1: 4-DOF robot agent training results for 5000 episodes at 300 steps with no human training

Next I looked at the results with human training included (Figure 4.2). I observed that the agent trained on human data, initially produces higher rewards and is able to reach 200 points before 1000 steps. This is due to our Q value neural network having been pre-trained by our human demonstration data set. However, the agent struggled with drop off where after a peak the agent produces bad results. Eventually the agent hits a max at a slightly earlier episode 4197. The drop off is likely caused by having too small of a human data set used for pre-training. The drop off could also have been caused by having too small a period between updates of our Q value neural network [20]. Having too frequent updates
can let our network over estimate Q values.

Figure 4.2: 4-DOF robot agent training results for 5000 episodes at 300 steps with human training

I also ran a training with 1000 steps during each episode. In a 1000 step environment it would be unlikely for the robot to hit maximum, since the ideal case can reach the target in less than 300 steps [22]. In (Figure 4.3) I can see our robot agent achieved a maximum of 452 point out of 1000 at episode 4001. Using random exploration, our algorithm now provided less immediate feedback on an episode basis. Our agent is more likely to oscillate between positive and negative rewards.

Figure 4.3: 4-DOF robot agent training results for 5000 episodes at 300 steps with no human training

I trained the human demonstration pre-trained agent 1000 steps per episode, for 5000 episodes (Figure 4.4). Our human trained agent hits 600 points at episode 1456. Since, our
agent was pre-trained by human demonstration data, selecting actions that were predicted by our Q value network would yield more accurate steps than random exploration [20]. I also suffered less from drop off since, our neural network was able to see more examples in training.

Figure 4.4: 4-DOF robot agent training results for 5000 episodes at 300 steps with human training

4.2 Execution Examples

Once I have a trained intelligent agent I can ask the non-human trained robot agent to attempt to reach a random target position from a random starting position. Below are five prime examples of the robot arm position error where I have removed the on target criteria to see how close the agent can get. In (Figure 4.5) the robot agent reached a minimum position error of 0.58 cm in 183 steps. I can also see that the position error did not decrease linearly and has a few inflection points which points towards the inaccuracies in actions selected which slow down the progress. In (Figure 4.6) the robot agent reached a minimum position error of 2.3 cm in 129 steps. I can also see that the position error decreased more linearly and in few steps then iteration 1. In (Figure 4.7) the robot agent reached a minimum position error of 0.63 cm in 141 steps. I can also see that the position error decreased more linearly. In (Figure 4.8) the robot agent reached a minimum position error of 0.13 cm. In the plot I see the error decayed very quickly, then switched to a slow regime to the end.
For (Figure 4.9) our robot agent reached a minimum position of 1.48 cm. In each of these examples, the agent was able to reach very close to the target position. However, looking at the different error paths, I see linearity breaking behavior. This is caused by the agent adopting a preference for minimizing the error by moving $\theta_1$ first. For the examples where the error decayed really quickly, then shifted to slow, is where the agent almost moved $\theta_1$ to the limit, then moved the other axes.

Figure 4.5: Robot Execution 1, minimum distance: 0.58 cm

Figure 4.6: Robot Execution 2, minimum distance: 2.3 cm
After training our human demonstration trained robot agent I tasked the agent with moving to a random target position from a random starting position. In (Figure 4.10) our
agent reached a minimum position error of 2.4 cm. However, I can see that the error had a more linear slope. The minimum was also found in fewer steps, 93 steps. In (Figure 4.11) I reached a minimum of 4.3 cm but also saw a error path the indicated a preference for moving $\theta_1$. In (Figure 4.12) I reached a minimum position of 2.6 cm but produced a very linear path. In (Figure 4.13) I reached a minimum position of 1.8 cm. Our error path was linear but on the small scale there was a lot of jittering. In (Figure 4.14) I reached a minimum position of 0.9 cm. Our error path was linear. Our human trained agent was able to produce more linear error paths. This linearity came from our human example pre-training.

Figure 4.10: Human Trained Robot Execution 1, minimum distance: 2.4 cm

Figure 4.11: Human Trained Robot Execution 2, minimum distance: 4.3 cm
Figure 4.12: Human Trained Robot Execution 3, minimum distance: 2.6 cm

Figure 4.13: Human Trained Robot Execution 4, minimum distance: 1.8 cm

Figure 4.14: Human Trained Robot Execution 5, minimum distance: 0.9 cm
4.3 Execution Accuracies

Once our non-human trained intelligent agent was trained, I tasked it with 1000 moves to random target positions from random starting position task executions. I then recorded the least distance to the target I were able to achieve each execution. In (Figure 4.15) I can see that our robot arm agent was able to reach within 5 cm of our target position 48.2% of the time. When I decreased the radius to 1 cm, I saw that our robot agent is able to reach within 1 cm of our target position 11.2% of the time (Figure 4.16). Our agent learned to move towards the target. However, our agent was not able to get on target consistently. However, considering the size of our state-action space, this is far from a random result.

Figure 4.15: 1000 Random Target Position Executions, 5cm Bins
I then tasked our human trained learning agent to execute the same 1000 moves to random target positions from random starting position tasks. I observed that our human trained learning agent reached within 5 cm of our target position 92.7% of the time (Figure 4.17). However, when counting how many executions reached within 1 cm, our human trained agent only achieved 4.4% (Figure 4.18). Our human trained agent greatly benefited from our human pre-training. The discrepancy in our ability to get within 5 cm but not 1 cm indicates that there is error present in our forward kinematics or in our human recording that our Q value network is not able to absorb and correct for. For example, as I explained in Chapter 3, a small change in our DH parameters, generated a large change in our end effector position. Also, I controlled out robot by hand with the servos off, so error could also be introduced in our recordings.

Figure 4.16: 1000 Random Target Position Executions with Human Training, 5cm Bins
4.4 Comparisons to Prior Work

4.4.1 Robot Arm

In our work I moved out of simulation to hardware. Unlike [5] and [1] I had success in our implementation on a real robot arm. The researches utilized external images to calculate the
position of their end effector position. This allowed them to create a model free robot agent focused only on 3 joint variables. However, I utilized joint sensors to calculate end effector position via forward kinematics. This constrained us to a single robot configuration. But since I hard coded our robot arm configuration parameters it allowed us to easily transition out of simulation into hardware.

4.4.2 Neural Network

Since I aren’t using images as feedback, I don’t need to use convolution in our neural network. Our Neural Network is a dueling double deep Q network that only takes about an hour to complete training.

4.4.3 Policy and Rewards

I rewarded the robot arm after every step. Since I are utilizing one degree motions in a very large state action space, I saw immediate rewards that produced an agent that could reach the target. If I were dealing with full trajectories, reward only on completed target reaching behavior would make more sense.

In [7] complex policies are learned via soft Q learning. They even allowed for the composition of policies. Like [2] I used a greedy policy.

4.4.4 Human Involvement

In our experiment, human demonstrations were generated by a human physically moving the robot similar to [9] where a 7-DOF robot arm was physically moved. These examples were stored as experiences in our replay memory, and utilized to training our neural network. I differed from [8] which focused on human selection of proposed trajectories. And these results differed from [10] which utilized policies to define human like behavior.
5.1 Conclusion

Our results show that a trained robot agent is able to successfully reach a given target position by selecting an action given by a Q value prediction from a dueling deep q value network.

The non-human robot agent is able to reach the target within 5 cm 48.2% of the time. This figure is improved by the inclusion of human training, where the human trained agent is able to reach within 5 cm of target 92.7% of the time. However, when reduced to a radius of 1 cm, both models showed decreased success; robot agent 11.2% and human agent 4.4%. Both of these agents struggled close to target because of our errors in forward kinematic calculation. Our non-human robot agent was able to compensate for these errors a little bit from their random exploration examples. However, our human demonstration data set was generated on a real robot arm by hand. And the manual recordings allowed for a lot of error to be introduced. I also see that our human trained agent executes on target position in fewer steps than our non human trained agent. Also, the error plots of our human trained agent are much more linear than our non human train robot agent. For training, I showed that the inclusion of human training data accelerated and improved the intelligent agent’s training. When deployed on a physical robot arm and combined with an object detection system, our robot agent could be utilized in a pick and place or peg insertion application.
5.2 Future Work

5.2.1 Robot Arm

During the final phase of this research our robot arm failed. Thus I were unable to train our robot agent utilizing our physical robot arm. Training on a physical robot arm would have allowed us to absorb the idiosyncrasies, errors, and perturbations present on our robot arm.

5.2.2 Model Free Agent

In our experiment I depended on our internal sensor positions to calculate the end effector position via forward kinematic calculations. There is error introduced in our end effector position by the accuracy of our DH parameters and rounding errors in our computation. Furthermore, by relying on forward kinematics hard coding I locked our agent to one robot arm and configuration.

I could use external position sensors to measure the end effector position which would allow us to create a model free learning agent which would increase the accuracy and precision of our experiment.

5.2.3 Reward and Policy

In our experiment I utilized a simple reward function which would reward or punish after every step. I could add complexity to our reward function by reward after a whole trajectory or only when reaching the target position.

Our policy is also simple. The greedy policy only takes the maximum Q value available. However, I could modify our policy to be more dynamic based upon current position.

5.2.4 Data Set

I were able to create a data set of 10,000 step examples. However, I found our data set was still too small compared to our state-action. I would like to record hundreds of thousands
of examples.

5.2.5 Natural Language Processing

The agent learns to select actions based upon deep Q learning from example steps. However, it would be a novel idea to add natural language processing to the action selection. This would allow a human to control the robot arm via English instruction set.
Appendices
Tossen Robotics PhantomX Reactor Robot Arm Kit

Nvidia GTX 1080Ti

Python 3.7

Tensorflow 2.0
class RobotArmEnvV2:

    def __init__(self):
        self.set_link_properties([1, 1, 1, 1])
        self.set_increment_rate(0.0174533)
        self.p = 0
        self.r = 0
        self.re = []
        self.target_pos = self.generate_random_pos()
        self.current_error = -math.inf
        self.viewer = None

    def set_link_properties(self, links):
        self.links = links
        self.n_links = len(self.links)
        self.min_theta1 = math.radians(0)
        self.min_theta2 = math.radians(50)
        self.min_theta3 = math.radians(75)
        self.min_theta4 = math.radians(20)
        self.max_theta1 = math.radians(300)
        self.max_theta2 = math.radians(130)
        self.max_theta3 = math.radians(150)
        self.max_theta4 = math.radians(210)
        self.theta = self.generate_random_angle()
        self.max_length = sum(self.links)

    def set_increment_rate(self, rate):
        self.rate = rate

    def forward_kinematics(self, theta):
        theta_1 = math.degrees(theta[0])
        theta_2 = math.degrees(theta[1])
        theta_3 = math.degrees(theta[2])
\[ \theta_4 = \text{math.degrees}(\theta(3)) \]

\[ a_1 = 108 \quad \# \text{Length of link 1} \]
\[ a_2 = 146 \quad \# \text{Length of link 2} \]
\[ a_3 = 40 \quad \# \text{Length of link 3} \]
\[ a_4 = 146 \]
\[ a_5 = 160 \quad \# \text{Length of link 4} \]

d_h_table = np.array([[np.deg2rad(\theta(1)), np.deg2rad(-90), 0, a_1],  
[0, np.deg2rad(90), 0, -a_3],  
[np.deg2rad(\theta(3) = 175), 0, a_4, 0],  
[np.deg2rad(\theta(4) = 105), 0, a_5, 0]])

i = 0

homgen_0_1 = np.array([[-np.sin(d_h_table[i, 0]) + np.cos(d_h_table[i, 0]) * np.cos(d_h_table[i, 1]) * np.sin(d_h_table[i, 2]), np.cos(d_h_table[i, 0]) * np.cos(d_h_table[i, 1]) - np.sin(d_h_table[i, 2]) * np.cos(d_h_table[i, 0]), np.cos(d_h_table[i, 1]) * np.sin(d_h_table[i, 2]) + np.sin(d_h_table[i, 0]) * np.cos(d_h_table[i, 1]) - np.sin(d_h_table[i, 2]) * np.cos(d_h_table[i, 0]), d_h_table[i, 2] * np.cos(d_h_table[i, 0]),  
[0, np.cos(d_h_table[i, 1]), np.sin(d_h_table[i, 1]), d_h_table[i, 3]],  
[0, 0, 0, 1]])

i = 1

homgen_1_2 = np.array([[-np.sin(d_h_table[i, 0]) + np.cos(d_h_table[i, 0]) * np.cos(d_h_table[i, 1]) * np.sin(d_h_table[i, 2]), np.cos(d_h_table[i, 0]) * np.cos(d_h_table[i, 1]) - np.sin(d_h_table[i, 2]) * np.cos(d_h_table[i, 0]), np.cos(d_h_table[i, 1]) * np.sin(d_h_table[i, 2]) + np.sin(d_h_table[i, 0]) * np.cos(d_h_table[i, 1]) - np.sin(d_h_table[i, 2]) * np.cos(d_h_table[i, 0]), d_h_table[i, 2] * np.cos(d_h_table[i, 0]),  
[0, np.cos(d_h_table[i, 1]), np.sin(d_h_table[i, 1]), d_h_table[i, 3]],  
[0, 0, 0, 1]])
\[
\begin{align*}
&d_{\cdot \cdot \cdot \cdot \cdot \cdot \cdot} = \cos(d_{\cdot \cdot \cdot \cdot \cdot \cdot \cdot}) \cdot \sin(d_{\cdot \cdot \cdot \cdot \cdot \cdot \cdot}) \cdot \sin(d_{\cdot \cdot \cdot \cdot \cdot \cdot \cdot})
\end{align*}
\]

```python
def generate_random_angle(self):
    theta = np.zeros(self.n_links)
    theta[0] = math.radians(random.uniform(50, 250))
    theta[1] = math.radians(90)
    theta[2] = math.radians(100)
    theta[3] = math.radians(random.uniform(50, 120))
    return theta

def generate_random_pos(self):
    theta = self.generate_random_angle()
    self.theta = theta
    self.p = random.uniform(0, 300)
    self.r = random.uniform(15, 30)
    pos = np.array((self.r * math.cos(math.radians(self.p)), self.r * math.sin(math.radians(self.p)), 0))
    return pos

def step(self, action):
    a = list(self.action[action])
    for n in range(len(a)):
        if a[n] == '2':
            self.theta[n] = self.theta[n] - self.rate
        elif a[n] == '1':
            self.theta[n] = self.theta[n] + self.rate
        else:
            continue
    P = self.forward_kinematics(self.theta)
    tip_pos = P
    pos = tip_pos
    target = self.target
    pt = self.p
    pp = math.degrees(self.theta[0])
    diffp = math.radians(abs(pp-pt))
    rt = math.sqrt(target[1]**2 + target[0]**2)
    rp = math.sqrt(pos[1]**2 + pos[0]**2)
    diffh = min(rt, rp)*diffp
    diffz = abs(pos[2] - target[2])
    diffd = abs(diffh**2 + diffp**2)
    distance_error = diffh + diffp
    reward = 0
    if distance_error >= self.current_error:
        reward = -1
    if distance_error <= self.current_error:
        reward = 1
    if len(self.re) == 10:
```

```python
def reset(self):
    self.target_pos = self.generate_random_pos()
    self.current_score = 0
    self.re = []
    observation = np.hstack((self.target_pos, self.theta))
    return observation

class ReplayMemory:
    def __init__(self, max_size):
        self.buffer = np.empty(max_size, dtype=np.object)
        self.max_size = max_size
        self.index = 0
        self.size = 0
    def append(self, obj):
        self.buffer[self.index] = obj
        self.size = min(self.size + 1, self.max_size)
        self.index = (self.index + 1) % self.max_size
    def sample(self, batch_size):
        indices = np.random.randint(self.size, size=batch_size)
        return self.buffer[indices]

class QNetwork(tf.Module):
    def __init__(self, lr=1e-3, input_shape=[7], n_outputs=81):
        super(QNetwork, self).__init__()
        self.optimizer = keras.optimizers.Adam(lr=1e-3)
        self.loss = keras.losses.Huber()
        self.model = self.get_model()
    def get_model(self):
        input_shape = [7]
        n_outputs = 81
        K = keras.backend
        input_states = keras.layers.Input(shape=[7])
        hidden1 = keras.layers.Dense(128, activation="elu") (input_states)
        hidden2 = keras.layers.Dense(128, activation="elu") (hidden1)
```

hidden3 = keras.layers.Dense(128, activation="elu")(hidden2)
hidden4 = keras.layers.Dense(128, activation="elu")(hidden3)
hidden5 = keras.layers.Dense(128, activation="elu")(hidden4)
state_values = keras.layers.Dense(1)(hidden5)
raw_advantages = keras.layers.Dense(n_outputs)(hidden5)
advantages = raw_advantages - K.max(raw_advantages, axis=1, keepdims=True)
Q_values = state_values + advantages
model = keras.models.Model(inputs=[input_states], outputs=[Q_values])
return model

class DQNAgent(object):
    def __init__(self):
        self.epsilon = 1
        self.replay_memory = ReplayMemory(max_size=10000000)
        self.batch_size = 128
        self.discount_rate = 0.95
        self.network = QNetwork()
        self.model = self.network.model
        self.target = keras.models.clone(model=self.model)
        self.target.set_weights(model.get_weights())
        self.n_outputs = 81
        self.input_shape = [7]
        self.optimizer = self.network.optimizer
        self.loss_fn = self.network.loss
    def limit_check(self, df, env):
        theta = env.theta.copy()
        a = list(env.action[df['index']])
        for n in range(len(a)):
            if a[n] == '2':
                theta[n] = theta[n] - env.rate
            elif a[n] == '1':
                theta[n] = theta[n] + env.rate
            else:
                continue
        if theta[0] > 300 or theta[0] < 0:
            return True
        return False
    def limit_check2(self, df, env):
        theta = env.theta.copy()
        a = list(env.action[df])
        for n in range(len(a)):
            if a[n] == '2':
                theta[n] = theta[n] - env.rate
            elif a[n] == '1':
                theta[n] = theta[n] + env.rate
            else:
                continue
        if theta[0] > 300 or theta[0] < 0:
            return True
        return False
    def check_action(self, dfx, env):
collision = True
i = 0
dfx = dfx.sort_values(by=0, ascending=False)
dfx = dfx.reset_index()

while collision == True:
collision = self.limit.check(dfx.iloc[i], env)
if collision == True:
i += 1
return dfx.iloc[i]['index']

def random_action(self, ex):
    rans = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80]
    for r in ex:
        rans.pop(rans.index(r))
    return random.choice(rans)

def epsilon_greedy_policy(self, state, env, epsilon=0):
    if np.random.rand() < epsilon:
        collision = True
        actions = []
        while collision == True:
            action = self.random_action(actions)
            collision = self.limit.check2(action, env)
            if collision == True:
                actions.append(action)
        return int(action)
    else:
        Q_values = self.model.predict(state[np.newaxis])
        dfx = pd.DataFrame(Q_values[0])
        action = self.check_action(dfx, env)
        return int(action)

def sample_experiences(self, batch_size):
    indices = self.replay_memory.sample(batch_size)
    states, actions, rewards, next_states, dones = [
        np.array([experience[field_index] for experience in indices])
        for field_index in range(5)]
    return states, actions, rewards, next_states, dones

def play_one_step(self, env, state, epsilon):
    action = self.epsilon_greedy_policy(state, env, epsilon)
    next_state, reward, done, info = env.step(action)
    self.replay_memory.append((state, action, reward, next_state, done))
    return next_state, reward, done, info

def training_step(self, batch_size):
    experiences = self.sample_experiences(batch_size)
    states, actions, rewards, next_states, dones = experiences
    next_Q_values = self.model.predict(next_states)
best_next_actions = np.argmax(next_Q_values, axis=1)
next_mask = tf.one_hot(best_next_actions, n_outputs).numpy()
next_best_Q_values = (self.target.predict(
    next_states) * next_mask).sum(axis=1)
target_Q_values = (rewards +
    (1 - dones) * self.discount_rate * next_best_Q_values)
target_Q_values = target.Q_values.reshape(-1, 1)
mask = tf.one_hot(actions, n_outputs)
with tf.GradientTape() as tape:
    all_Q_values = self.model(states)
    Q_values = tf.reduce_sum(
        all_Q_values * mask, axis=1, keepdims=True)
    loss = tf.reduce_mean(self.loss_fn(target_Q_values, Q_values))
grads = tape.gradient(loss, self.model.trainable_variables)
self.optimizer.apply_gradients(
    zip(grads, self.model.trainable_variables))

env = RobotArmEnvV2()
np.random.seed(42)
tf.random.set_seed(42)
input_shape = [7]
n_outputs = 81
training_rewards = []
best_score = -1000

agent = DQNAgent()
for episode in range(5000):
    obs = env.reset()
    e_reward = 0

    for step in range(300):
        agent.epsilon = max(1 - episode / 4000, 0.01)
        obs, reward, done, info = agent.play_one_step(env, obs, agent.epsilon)
        e_reward += reward
        if done:
            break
    training_rewards.append(e_reward)

    if e_reward > best_score:
        best_weights = agent.model.get_weights()
        agent.model.save('non_human_model_43021727')
        best_score = e_reward
print("Episode {0}, Reward {1}, eps {2:.3f}".format(
    episode, e_reward, agent.epsilon), end='')

if episode > 500:
    agent.training_step(agent.batch_size)
if episode % 500 == 0 and episode >= 500:
    agent.target.set_weights(agent.model.get_weights())

agent.model.set_weights(best_weights)
REFERENCES


