Deep Reinforcement Learning in Physical Environments containing Continuous Action Spaces using a Prior Model with Applications to Robotic Control

Luke Taylor

University of Cape Town

Department of Mathematics and Applied Mathematics

Supervisor Dr. Jonathan Shock

August 2, 2018

Contents

List of Figures

List of Tables

Acknowledgement

I would like to thank my family for their endless support with all endeavours I have pursued over the years, encouraging me to follow my ambitions and to never stop dreaming.

Gratitude to the people of the Internet: Travis DeWolf for his tutorial on developing joint limb simulations using PyGame; Vikas Yadav for his implementation of the DDPG algorithm in PyTorch and Lilian Weng for her notes on Policy Gradient methods. Without their notes and code this thesis would have taken a considerable amount of time longer to complete.

Appreciation to everyone who has helped me proof read this thesis and giving me their valuable feedback: Dr. Jonathan Shock, my brother and the Heye family. I am grateful to everyone who has endured the considerable amount of complaining (especially AD and Sonia) due to my simulator doing weird things or my models not learning.

A special thanks to Dr. Jonathan Shock for granting me the possibility of pursuing further studies in Reinforcement Learning and his great mentorship in supervising this project; Our discussions relating to consciousness, intelligence and learning have been very stimulating.

On a final and serious note: Thank you to my mom for passing her German heritage onto me, giving me a passion for Weiss beer and the ability to build robots. I am of a strong belief that an AI uprising can be mitigated, if we teach robots the true values of life, one of which is being able to enjoy a cold one.

1 Introduction

1.1 Reinforcement Learning

Reinforcement Learning (RL) is a branch of Machine Learning that is concerned with learning behaviour, which is characterised by a sequence of decisions. The RL framework is simple: An agent $\mathcal{A}$ is situated in an environment $\mathcal{E}$. The agent observes a state s_t from the environment $\mathcal{E}$ upon which it takes an action a_t. On action completion the agent receives a reward r_t alongside a new environmental state s_{t+1}. This interaction between the agent $\mathcal{A}$ and environment $\mathcal{E}$ repeats, which is portrayed in figure 1.1.

The goal in RL is to learn behaviour for an agent $\mathcal{A}$, such as to maximise the cumulative reward received. Henceforth any objective the agent is ought to complete is encoded in the rewarding scheme. Surprisingly, complex behaviour can be learned in such a simple setup. An example is the RL system called AlphaGo [Sil+16] developed by Google DeepMind, that learned to play the game of Go, an ancient Chinese board game considered to be one of the hardest games for an Artificial Intelligence (AI) to master. The game of Go is simple by nature, as players place black or white stones onto a board in attempt to capture the opponent's pieces. The difficulty of the game arises from the number of possible board configurations, which is larger than the number of atoms in the observable universe, hence a reason why classic search based AI systems have failed at this game. AlphaGo managed to defeat the world title holder Mr Lee Sedol, showcasing the power that RL has to offer.

An exciting frontier of RL is the ability to develop new solutions. For example, DeepMind's AlphaGo system developed new winning strategies for the game of Go, albeit the game having been extensively studied for hundreds of years. It is natural to wonder: What else can RL aid mankind in search for new solutions?

1.2 Problem Motivation

The success of RL has mostly been validated in virtual environments. An active area of research is to transfer the success of RL to physical environments, where

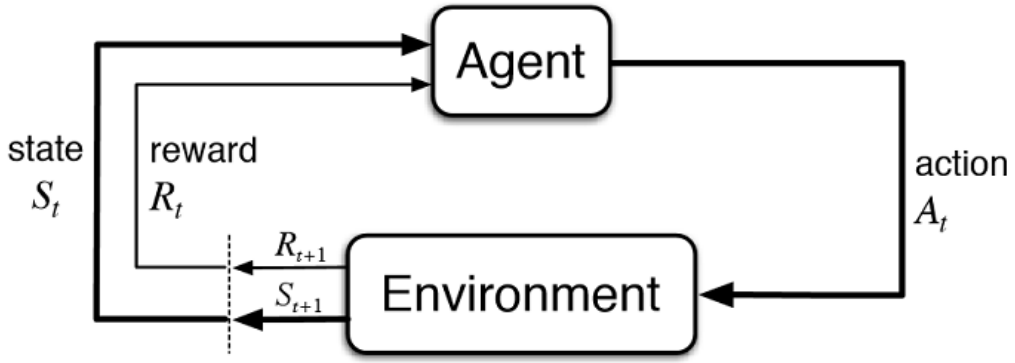

Fig. 1.1: Overview of RL framework, extracted from [SB+98]

the applications are vast: Learning to drive for self driving cars; Learning intelligent caring behaviour for human assisting robots or teaching robots in factories how to accomplish various manufacturing objectives. There are two fundamental reasons why the success of RL has not yet transferred to physical environments:

1. High-Sample complexity: Current RL algorithms require a large corpus of samples to learn meaningful behaviour. These samples are collected by the agent interacting with the environment: Apply an action, observe the new state and reward, apply another action, record the next state and reward and so on. In simulation, performing such interactions is quick. However, in a physical setting such as a robot learning to grasp an object, such interactions are painfully slow and take an extensive amount of time to complete.

2. Noisy non-stationary dynamics: The theoretical framework from which RL methods are derived is the so called Markov Decision Process (MDP), which mathematically represents the agent and the environment. In an MDP, it is assumed that the transition distribution $p(s'|s,a)$ is stationary i.e. the probability of transitioning from state s to state s' under action a is fixed throughout time. However, in many physical systems, such assumptions cannot be made, as the dynamics are time variant. In addition, many physical settings contain noise $\epsilon \sim \mathcal{N}$ sampled from unknown distributions that overlay the underlying transition dynamics.

The current approach of applying RL to physical settings, is to emulate the physical environment in simulation. It is in these emulated environments that the RL model learns. The learned model is then applied to the physical environment for deployment or for further fine tuning. A main problem with this approach is a social one: These simulated environments are usually constructed using expensive proprietary engines. An example is the Multi-Joint dynamics with Contact (MuJoCo) physics engine [Tod+12] which has been used in a variety of RL related research projects

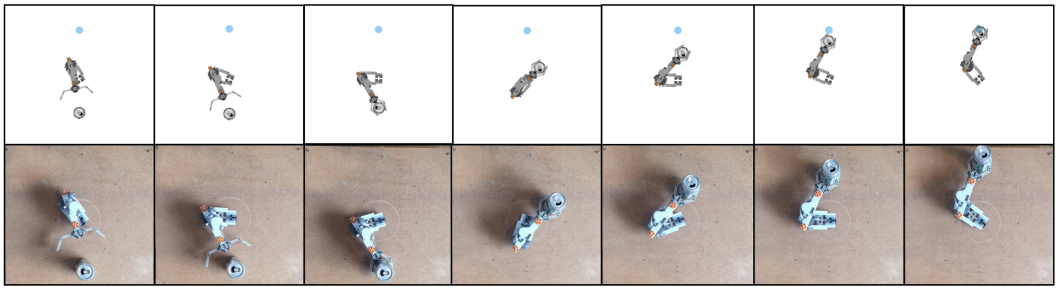

Fig. 1.2: A sequence of images extracted from video, of the robot performing the grasp task. The top row is the robot in simulation and the bottom row is the robot in the physical environment.

[Lil+15; Dua+16; Lev+16; Sch+15b], costing between 500 and 12000 dollars, depending on the license type. RL research applied to robotics is thus usually limited to research groups that have the financial budget to afford these engines, alongside the robots themselves, which raises the question: Can RL robotics research be conducted, without relying on expensive proprietary simulation and physics engines?

1.3 Problem Definition and Contribution

The problem investigated in this thesis is applying Reinforcement Learning to the task of robotic grasping defined in a physical setting with a continuous action space. The robotic grasping problem investigated is the following: A 2 degree of freedom robotic arm with a gripper is located on a 2 dimensional surface with the objective of having to locate an object, grasp it and move it to a variable goal position.

The means by which the problem is addressed is through the use of Deep RL methods, in which RL is coupled with Artificial Neural Networks, a type of non-linear function approximator; More specifically the Deep Deterministic Policy Gradient (DDPG) algorithm [Lil+15] was adopted. In addition, all training was performed in a custom built simulation using open-source tools, overcoming the prejudice that RL robotics research can only be conducted with expensive proprietary simulation and physics engines. Figure 1.2 portrays the built simulator alongside the physical robot, showcasing that the learned behaviour in the simulator is applicable to the physical environment.

1.4 Outline

The remaining chapters in this writeup build on each other, to give the necessary background knowledge to the reader to understand the methods adopted in tackling the problem outlined in the previous section. Although the content was written to

be as self contained as possible, it is suggested that the reader read the relevant literature to get a deeper understanding of the methods applied. The remaining chapters are structured as follows.

1. Preliminaries: This chapter introduces the fundamentals of Reinforcement Learning alongside Deep Learning, and how these two fields are combined, forming the study of Deep Reinforcement Learning.

2. Policy Gradients: In this chapter the RL paradigm of policy gradients is introduced, building up the necessary knowledge to understand the Deep Deterministic Policy Gradient algorithm.

3. Robotic Grasping: This chapter outlines how the robotic grasping problem was attempted, how the simulator was constructed and linked to the physical robot, how the DDPG algorithm was implemented and the results of benchmarking the learned model in virtual and physical experiments.

4. Conclusion: This chapter summarises the problem investigated, reflecting on potential reasons why certain methods failed and presents future research directions.

2 Preliminaries

2.1 Markov Decision Process

Markov Decision Processes (MDP) provides the necessary formalism for describing any Reinforcement Learning problem. A MDP is a collection of objects that characterises the environment, the agent and the objective. The environment is described by a set of possible states S. The agent can choose an action $a \in A$ from a set of available actions A. During a time-step t, the environment adopts a new state $s_t \in S$. The dynamics of the environment are characterised by the state transition distribution $P[S_{t+1} = s' | S_t = s, A_t = a]$. This distribution describes the probability of transitioning to a state s', conditioned on the environment being in the current state s and the agent choosing action a. This is formally described by the following definition.

Definition

A Markov Decision Process (MDP) is a six element tuple $< S, A, P, R, \mu, \gamma >$.

1. S: Is a finite set of states of the environment
2. A: Is a finite set of actions available to the agent
3. P: Is a state transition distribution. It defines the probability of moving to some state s' given the current state s and a chosen action a, $P[S_{t+1} = s' | S_t = s, A_t = a]$.
4. $R : S \times A \times S \to \mathbb{R}$: Is a reward function
5. μ: Is an initial state distribution, defining the probability of being in an initial state $s_0 \in S$.
6. γ: Is the discount factor, where $\gamma \in [0, 1]$.

(2.1)

For brevity, the initial state distribution μ and the discount factor γ are usually omitted in the literature (depending on the context of the problem being investi-

gated). The discount factor γ is introduced for a mathematical reason of bounding a sum, however is also used to influence the way an agent learns. The initial state distribution μ describes the probability of the environment starting in an initial state $s_0 \in S$. This is oftentimes introduced in the literature when mathematical analysis involving MDPs is carried out.

RL agents attempt to maximises the total discounted cumulative reward G_t. This is a summation of all rewards that the agent can receive when moving through the environment's space, before reaching an absorbing state s_T [1]. However, when the problem is of infinite-horizon (i.e. the trajectory of state observations and respective action choices $(s_0, a_0, s_1, a_1, \dots)$ is unbounded), then the summation of rewards is unbounded. The discount factor γ is introduced to bound this summation of rewards. This gives rise to the following definition of total discounted reward.

Definition

Return G_t is the total discounted cumulative reward from time-step t onwards which is defined by $G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$

(2.2)

The discount factor can also be used in finite-horizon problems to influence the way an agent learns. It establishes the means of weighting current and future rewards. When setting $\gamma \approx 0$, the agent places more emphasis on current rewards than rewards from the future. However, when $\gamma \approx 1$, the agent is more far-sighted and places equal weighting on rewards from the future. This mechanism allows for the manipulation of the total discounted reward, which in return changes the behaviour of the agent, as the agent will either strive for immediate returns, or pursue a more conservative approach that will strive for more returns from the future. The latter is generally of more interest; Being able to plan into the future by considering the consequences of the actions chosen now.

Definition

A state s_t at time t is said to satisfy the Markov property if $P[S_{t+1}|S_t] = P[S_{t+1}|S_1, \dots, S_t]$.

(2.3)

A fundamental property that characterises MDPs is the markovian property (formally defined above). When a stochastic process is markovian, it is considered to be

[1] This is a state which the agent cannot leave.

memoryless; All the relevant information is encoded in the current state of the process. The states preceding the current state are considered irrelevant. An example is the game of orthello; You arrive at a partially played game and continue playing without having to know the previous states of the board.

2.2 Policies

A policy is a mapping from states $s \in S$ to actions $a \in A$. This is the central problem that reinforcement learning attempts to solve, establishing algorithms that provide means of learning favourable policies. Policies are usually denotes by the symbol π and can be split into two categories: A deterministic and a stochastic policy. The deterministic policy is a mapping $\pi : S \to A$ and a stochastic policy is a mapping $\pi : S \to P(A)$ where $P(A)$ is the set of probability measures on the A. The choice of policy is dependent on the problem represented by the MDP. As an example, in the game of othello a deterministic policy would be preferred, as it is assumed that at each step in the game the player can play an optimal move. However, in the game of rock-paper-scissors, a stochastic policy would be optimal, as a deterministic policy could trivially be exploited.

The objective of a policy is to maximise the expected cumulative return G_t. The optimal policy is the policy that achieves the greatest expected cumulative return G_t and is denoted by π^*. In any given MDP, the existence of an optimal policy is guaranteed; There can exist more than one optimal policy in a given MDP.

There are three paradigms in RL for the construction of policies: Policy optimisation, dynamic programming and actor-critic methods. In the policy optimisation setting, the problem of constructing a policy is viewed as an optimisation problem. Here the policy is directly parameterised by some function π_θ containing n parameters $\theta \in \mathbb{R}^n$. In optimising the function π_θ, the parameters are shifted as to increase expected reward G_t. There are two ways of perturbing the parameters θ; Using evolutionary methods or using policy gradients methods. Evolutionary methods randomly perturb the parameters θ, measure performance of π_θ, and continue shifting weights in the direction associated with favourable performance. Evolutionary methods are simple to implement and work well in action spaces with low dimensionality n. Some examples of evolutionary methods are the cross-entropy method [SL06], the covariance matrix adaptation method [WP09] and the natural evolution strategies [Wie+08]. Policy gradient methods [Sut+00; Wil92; Kak02; Sch+15b], in contrast to evolutionary methods, apply changes to weights θ by approximating the gradient that shifts the parameters into a space of favourable performance. These methods are rather non-trivial to implement, however perform better in large action spaces, in

contrast to evolutionary methods. In dynamic programming, the objective is to learn value functions (discussed in more detail in the next section). A well established method of learning action-value functions is the Q-Learning algorithm [WD92]. Other algorithms for learning value functions are the policy iteration [SB+98] and value iteration algorithm [SB+98] which fall under the class of algorithms known as modified policy iteration. Actor-critic methods use both policy optimisation and dynamic programming in learning a policy. In this paradigm a parameterised policy π_θ is learned via policy optimisation (which is known as the actor), and value functions (learned via dynamic programming) are used to accelerate convergence and to reduce variance during optimisation (which is referred to as the critic). An example is the actor-critic method of [Lil+15] which is explained in more details in section 3.4.

The means by which a policy is learned, is classified into two categories: On-policy and off-policy. The distinction between these two paradigms, is how the exploration data for the learning of the policy is generated. In the on-policy paradigm, there is one distinct policy π that generates exploration data, and which itself gets optimised on that data. In contrast, the off-policy paradigm makes use of two distinct policies: A target policy π_T and a behavioural policy π_B. The target policy π_T is the policy that learns the optimal control whereas the behaviour policy π_B is the policy that generates the training data by exploring the environment. Henceforth, the off-policy paradigms generalises the on-policy paradigm with the on-policy paradigm having the target and behaviour policy being equivalent $\pi_T = \pi_B$.

2.3 Value Functions

Definition

The state-value function of a MDP is the expected return starting from a state s and following policy π thereafter.

$$V^\pi(s) = \mathop{\mathbb{E}}_{\pi}[G_t | S_t = s] \tag{2.4}$$

Definition

The action-value function of a MDP is the expected return starting from a state s, taking action a and following policy π thereafter.

$$Q^{\pi}(s,a) = \mathop{\mathbb{E}}_{\pi}[G_t | S_t = s, A_t = a] \tag{2.5}$$

The expected cumulative return of a policy π in a MDP $\mathcal{M}$ is expressed through the value functions $V^{\pi}(s)$ and $Q^{\pi}(s,a)$. Value functions are fundamental in reinforcement learning as they enable for the comparison of policies, indirect means of constructing a policy π and ways of reducing variance in policy gradient methods. In addition, their bellman equation representation (discussed shortly) form the foundation of many RL learning algorithms.

The state-value function $V^{\pi}(s)$ defines the expected return when starting in a state s and following the policy π thereafter. The definition of the action-value function $Q^{\pi}(s,a)$ is similar, however here it is assumed that the environment is in state s and an action a (this is any arbitrary legal action, which can be different to the action chosen by the policy $\pi(s)$) is chosen before the policy π is followed.

Definition

The bellman equations for the state-value and action-value function are defined as follows.

$$V^{\pi}(s) = \sum_{a \in A} \pi(a|s) \sum_{s' \in S} p(s'|s,a)\left[r(s',a,s) + \gamma V^{\pi}(s')\right] \tag{2.6}$$

$$Q^{\pi}(s,a) = \sum_{s' \in S} p(s'|s,a)\left[r(s',a,s) + \gamma V^{\pi}(s')\right] \tag{2.7}$$

The bellman equations define a recursive relation between the value function in one state to the value functions on previous states. This is analogous to the dynamic programming paradigm: Being able to solve a problem by knowing the solutions to its subproblems. Having defined the bellman relation for the state-value function $V^{\pi}(s)$ and the action-value function $Q^{\pi}(s,a)$, the following relation between these two functions can be observed.

$$V^{\pi}(s) = \sum_{a \in A} \pi(a|s) Q^{\pi}(s,a) \tag{2.8}$$

This relation showcases that one can solve the state-value function V^π, action-value function Q^π or the policy π, if one knows the values of the other two functions (as this forms a system of linear equations). In addition, the definition of these two value functions gives rise to a theorem know as the policy improvement theorem 2.9.

Policy Improvement Theorem

Let $\pi, \pi' \in \prod_{\mathcal{M}}$ be deterministic policies applicable to MDP $\mathcal{M}$.

$$Q^\pi(s, \pi'(s)) \geq V^\pi(s) \; \forall s \in S \implies V^{\pi'}(s) \geq V^\pi(s) \tag{2.9}$$

Definition

Optimal value functions are the maximum value functions with respect to a policy. The maximum state-value function is defined as.

$$V^*(s) = \max_\pi V^\pi(s) \tag{2.10}$$

The maximum action-value function is defined as

$$Q^*(s, a) = \max_\pi Q^\pi(s, a) \tag{2.11}$$

Definition

A policy π^* is optimal if for all policies $\pi \in \prod_{\mathcal{M}}$ in a given MDP $\mathcal{M}$ the following is satisfied $V^{\pi^*} \geq V^\pi$.

(2.12)

Optimal value functions define the maximum amount of total expected reward that can be received. The maximum state-value function V^* defines the maximum amount of total cumulative reward from a state $s \in S$ onwards and the maximum action-value function Q^π defines the maximum amount of total cumulative reward from a state $s \in S$ taking action $a \in A$ onwards. As per the definition above, these optimal value functions are expressed in terms of a maximising policy π. Henceforth, an optimal policy π^* (as defined in definition 2.12) can trivially be extracted given the optimal action-value function as follows: $\pi^*(s) = \max_{a \in A} Q^*(s, a)$

Having defined the bellman equation for the state-value function V^π and the policy improvement theorem, one can derive the established policy iteration algorithm

for learning an optimal policy π^* via the construction of a sequence of state-value functions $\{V^{\pi_i}\}_i$ that converge to an optimal state-value function V^* in the limit $i \to \infty$. This algorithm starts with an arbitrary policy π_0 for which its respective state-value function V^{π_0} is found via a method called policy evaluation, which is derived from the bellman equation 2.6. After this construction, the policy π_0 is improved on using $\pi(s) \leftarrow \underset{a \in A(s)}{\text{argmax}} Q^{\pi}(s,a)$ generating a new policy π_1. This procedure is continued until convergence is achieved. See [SB+98] for an detailed construction of the algorithm alongside convergence proofs. Outlining a summary of the algorithm, serves the purpose of motivating the importance of the definition of the value functions and the relations and theorems that are derived using them.

Definition

The bellman optimality equations for the state-value and action-value function are defined as follows.

$$\begin{aligned} V^*(s) &= \max_a \mathbb{E}[R_{t+1} + \gamma V^*(S_{t+1}) | S_t = s, A_t = a] \\ &= \max_a \sum_{s' \in S} p(s'|s,a)\left[r(s',a,s) + \gamma V^*(s')\right] \end{aligned} \tag{2.13}$$

$$\begin{aligned} Q^*(s,a) &= \mathbb{E}[R_{t+1} + \gamma \underset{a'}{\text{max}} Q^*(S_{t+1}, a') | S_t = s, A_t = a] \\ &= \sum_{s' \in S} p(s'|s,a)\left[r(s',a,s) + \gamma \underset{a'}{\text{max}} Q^*(s',a')\right] \end{aligned} \tag{2.14}$$

On a closing note, using the definition of the bellman equations 2.6 and 2.7 and the definition of the optimal value functions, one arrives at the bellman optimality equations outlined above. The importance of these equations, is that they are expressed independent of a policy π. Solving these equations enables the extraction of the optimal policy π^*. These equations root the foundation of the value iteration algorithm, which is another way of deriving a policy π in a model-based setting using value functions.

2.4 Deep Learning

Deep Learning (DL) is a branch of Machine Learning (ML), the study concerned with learning from data. DL has revolutionised the field of ML by pushing the frontiers with regards to maintaining state of the art performance across various domains, such as speech recognition [Amo+16b], translation [Bah+14] and image detection

[Ren+15]. When talking about DL, one usually refers to the nonlinear function approximator known as the Artificial Neural Network (ANN).

In the supervised learning paradigm we have a set of N labeled data items $X = \{(x_i, y_i)\}_{1...N}$ for which we want to learn a mapping $f : x \to y$. ANNs provide the means of learning such a mapping. In its simplest form, when talking about ANNs, one usually refers to a Directed Acyclic Graphs (DAG), where every node represent an artificial neuron and every edge represents a weighting between neurons. This setup is referred to as a feedforward neural network. There are other types of abstractions of this setup, for example the Convolutional Neural Networks [LeC+98] and Recurrent Neural Networks [Gro13], however they remain out of scope for this writeup.

The feedforward neural network consists of three types of layers, an input layer, a hidden layer and an output layer. Every layer consists of a column of nodes. There is only one input layer and one output layer, with a variable number of hidden layers. In reference to DL, one usually refers to ANNs with multiple hidden layers. The more hidden layers there are, the deeper the network, allowing for more expressive associations to be formed; This is a trend that has been observed in the development of state of the art image recognition systems, benchmarked on the ImageNet dataset [Den+09]. In 2012 the AlexNet [Kri+12] consisting of 8 layers had a top-5 error rate of 15.3%; In 2014 the GoogLeNet [Sze+15] was built out of 22 layers and scored a 6.7% top-5 error rate and in 2015, the Resnet [He+16] developed out of 152 layers, scored a top-5 error rate of 2.25%.

When talking about ANNs, one usually refers to specific modules which are groupings of neurons. In the case of the feedforward networks, one refers to columns of neurons as fully connected layers. [2] Every neuron in layer i is connected to every neuron in layer $i - 1$. [3]

The output of every node is computed as follows: Multiply the output from every node in the previous layer by an associated weight, summate all these terms and

[2] In the case of Convolutional Neural Network, one gets convolutions layers and max pooling layers; Or in Recurrent Neural Networks on get modules called LSTMs, hence the motivation of grouping neurons by the type of computation that they perform.

[3] If no connection is to be established, one could set the weight of an edge to 0; However, these weights are never manually entered, yet rather learnt.

finally pass this summation through a non-linear function (known as a activation function). This is mathematically formalised as follows.

$$v_j^i = f_j^i(\underbrace{\sum_{k=1}^{n_{i-1}} w_{jk}^i v_k^{i-1} + b_j^i}_{z_j^i}) \tag{2.15}$$

Here v_j^i is the output of node j in layer i; f_j^i is the activation function used at this respective node; w_{jk}^i is the weighting of the edge between the k^{th} node in layer $i-1$ and the j^{th} node in layer i; v_k^{i-1} is the output of node k from layer $i-1$; b_j^i is a bias term and n_{i-1} is the amount of nodes contained in layer $i-1$.

The bias b_j^i term is useful, in that it allows for the shifting of the activation function along the x-axis; Which may or may not be critical in learning an expressive mapping. In summary, it allows for a more expressive model, which would not be possible if it were not for the weight bias. It is useful to absorb the term b_j^i into the summation. This can be done by adding an extra node to every layer (except the output layer) defined by $v_0^{i-1} = 1$ for $1 \leq i \leq m$ (where 0 is the input layer and m is the output layer) where the extra edge becomes the bias $b_j^i = w_{j0}^i$. Henceforth equation 2.15 can be rewritten as.

$$v_j^i = f_j^i(\underbrace{\sum_{k=0}^{n_{i-1}} w_{jk}^i v_k^{i-1}}_{z_j^i}) \tag{2.16}$$

The literature on activation functions f_j^i is vast; In the initial onset of ANNs, activation functions such as the sigmoid function [4] defined by $S(x) = \frac{e^x}{e^x+1}$, were a popular choice. However, with the development of deeper networks, the vanishing gradient problem became an issue [Hoc+01], which required activation functions that did not suffer from this problem. [5] An activation function that solved this problem, and which is a useful goto, is the Rectified Linear Unit (ReLu) activation function; This function is defined by $f(x) = max(0, x)$. This activation functions was one of the reasons for the success of the iconic AlexNet paper [Kri+12].

[4] Interestingly, publishing on the properties of a single activation function [HM95] at a respected venue was possible back in the 90s; This showcases how fast the field of Deep Learning has grown over the years!

[5] The vanishing gradient occurs due to the construction of the backpropagation algorithm: Gradients slowly dissipate for further updates from the output layer.

A fundamental component, and which makes DL a learning paradigm, is the method of finding the values for the weights w^i_{jk}, which is known as backpropagation. Most applications in which non-linear optimisation is performed, is done through an iterative procedure: For example, the parameters of GLMs are found via the Iteratively Reweighted Least Squares algorithm, or certain non-linear ODEs are solved via the Newton-Kontorovich method. The same applies to ANNs, for which the iterative procedure is gradient descent. In order to perform gradient descent in the context of ANNs, one needs a cost function $\mathcal{L}(t_i, y_i)$ that defines the error between the prediction of the network t_i and the actual label that it should have predicted y_i. There is a wide array of possible cost functions $\mathcal{L}(t_i, y_i)$ available; A classic example is the mean squared cost function defined by $\mathcal{L}(t_i, y_i) = \frac{1}{2}(t_i - y_i)^2$. The total error on a dataset X of N elements is then defined by.

$$E(X) = \sum_{i=1}^{N} \mathcal{L}(t_i, y_i) \tag{2.17}$$

The objective is to learn weights w^i_{jk} that minimise the specified cost function. This is achieved by updating all parameters w^i_{jk} in the direction that minimise the error $E(X)$, which is achieved with the following update rule.

$$w^i_{jk} \leftarrow w^i_{jk} - \eta \frac{\partial E(X)}{\partial w^i_{jk}} \tag{2.18}$$

Here $\eta > 0$ is a hyper-parameter known as the learning rate, controlling how fast the weights are to be updated. A too low learning rate can substantially slow down the learning process, whereas a too high learning rate could overshoot a minimum.

$$\begin{aligned} \frac{\partial E(X)}{\partial w^i_{jk}} &= \frac{\partial}{\partial w^i_{jk}} \sum_{i=1}^{N} \mathcal{L}(t_i, y_i) \\ &= \sum_{i=1}^{N} \frac{\partial}{\partial w^i_{jk}} \mathcal{L}(t_i, y_i) \end{aligned} \tag{2.19}$$

Note that the gradient of the error $\frac{\partial E(X)}{\partial w^i_{jk}}$ is the summation of all the gradients $\frac{\partial \mathcal{L}}{\partial w^i_{jk}}$, which is shown in the equations 2.19. Henceforth, to be able to perform update rule 2.18 one has to be able to compute $\frac{\partial \mathcal{L}}{\partial w^i_{jk}}$. The means by which $\frac{\partial \mathcal{L}}{\partial w^i_{jk}}$ is computed, is accomplished via backpropagation.

2.4.1 Backpropagation Derivation

As a reminder, we make the following definition of the multivariate chain rule. Note that in this definition, if we set $m = 1$ and $n = 1$, we get the usual chain rule that one is accustomed to from first year calculus, which is $\frac{d}{dx}(f \circ g) = \frac{df}{dg}\frac{dg}{dx}$.

Multivariate Chain Rule

Let $f : \mathbb{R}^m \to \mathbb{R}$ and $g : \mathbb{R}^n \to \mathbb{R}^m$ then

$$\frac{d}{dx_i}(f \circ g) = \sum_{l=1}^{m} \frac{\partial f}{\partial g_l}\frac{\partial g_l}{\partial x_i} \tag{2.20}$$

In addition, we recall from the last section, that the computation that every node in the network performs (apart from the nodes in the input layer) is the following.

Nodal Computation

v_j^i is the output of node j in layer i where f_j^i is the activation applied at this respective node.

$$v_j^i = f_j^i(z_j^i) \tag{2.21}$$

z_j^i is the product sum (including bias b_j^i) of node j in layer i. w_{jk}^i is the weight between node k from layer $i-1$ and node j from layer i and n_{i-1} is the amount of nodes in layer $i-1$.

$$z_j^i = \sum_{k=0}^{n_{i-1}} w_{jk}^i v_k^{i-1} \tag{2.22}$$

Consider an ANN with m layers, here the 0^{th} layer is the input layer and the m^{th} layer is the output layer. All layers in between are hidden. In order to compute $\frac{\partial \mathcal{L}}{\partial w_{ij}^k}$, we expand expand using the chain rule.

$$\frac{\partial \mathcal{L}}{\partial w_{ij}^k} = \frac{\partial \mathcal{L}}{\partial v_i^k}\frac{\partial v_i^k}{\partial z_i^k}\frac{\partial z_i^k}{\partial w_{ij}^k} \tag{2.23}$$

The partial derivate $\frac{\partial z_i^k}{\partial w_{ij}^k}$ is computed as follows.

$$\begin{aligned}\frac{\partial z_i^k}{\partial w_{ij}^k} &= \frac{\partial}{\partial w_{ij}^k} \sum_{l=0}^{n_{k-1}} w_{il}^k v_l^{k-1} \\ &= v_j^{k-1}\end{aligned} \tag{2.24}$$

The partial derivate $\frac{\partial v_i^k}{\partial z_i^k}$ is dependent on the chosen activation function. As an example, if we were to choose the sigmoid function as the activation function, then $f_i^k(z_i^k) = S(z_i^k)$ for which $\frac{\partial v_i^k}{\partial z_i^k}(z_i^k) = S(z_i^k)(1 - S(z_i^k))$.

The partial derivative $\frac{\partial \mathcal{L}}{\partial v_i^k}$ is to be considered for two cases: For the output layer and for the hidden layers. For a neuron in the output layer (i.e. when k = m), the partial derivative $\frac{\partial \mathcal{L}}{\partial v_i^k}$ can directly be computed, given the definition of the cost function $\mathcal{L}$. As an example, consider the mean squared error cost function $\mathcal{L}(t_i, y_i) = \frac{1}{2}(t_i - y_i)^2$, for which $\frac{\partial \mathcal{L}}{\partial v_i^k} = t_i - y_i$ (if we consider the output layer to only have one neuron $v_1^m = t_1$).

If v_i^k is is a neuron in the hidden layer (i.e. $1 \leq k < m$) then we can consider the cost function $\mathcal{L}$ to be a function of all product sums z_i^{k+1} for $i = 1, \ldots, n_{k+1}$.

$$\mathcal{L} = \mathcal{L}(z_1^{k+1}(v_1^k, v_2^k, \ldots, v_{n_k}^k), z_2^{k+1}(v_1^k, v_2^k, \ldots, v_{n_k}^k), \ldots, z_{n_{k+1}}^{k+1}(v_1^k, v_2^k, \ldots, v_{n_k}^k))$$

Using this form we can apply the multivariate chain rule 2.20 to compute $\frac{\partial \mathcal{L}}{\partial v_i^k}$.

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial v_i^k} &= \sum_{l=0}^{n_{k+1}} \frac{\partial \mathcal{L}}{\partial z_l^{k+1}} \frac{\partial z_l^{k+1}}{\partial v_i^k} \\ &= \sum_{l=0}^{n_{k+1}} \frac{\partial \mathcal{L}}{\partial v_l^{k+1}} \frac{\partial v_l^{k+1}}{\partial z_l^{k+1}} w_{li}^{k+1}\end{aligned} \tag{2.25}$$

It can be observed that the gradients in layer k can only be computed knowing the gradients of layer $k+1$, hence the name backpropagation, as we propagate the gradients backwards. The partial derivative $\frac{\partial \mathcal{L}}{\partial v_i^k}$ can be summarised as follows.

$$\frac{\partial \mathcal{L}}{\partial w_{ij}^k} = \begin{cases} v_j^{k-1} \frac{\partial v_i^k}{\partial z_i^k} \frac{\partial \mathcal{L}}{\partial v_i^k}, & \text{if } k = m. \\ v_j^{k-1} \frac{\partial v_i^k}{\partial z_i^k} \sum_{l=0}^{n_{k+1}} \frac{\partial \mathcal{L}}{\partial v_l^{k+1}} \frac{\partial v_l^{k+1}}{\partial z_l^{k+1}} w_{li}^{k+1}, & \text{otherwise.} \end{cases} \tag{2.26}$$

2.5 Deep Reinforcement Learning

Deep Reinforcement Learning is the field of study in which ANNs are applied to RL. The first work demonstrating the use of ANNs in RL, dates back to the 1990s [Tes95]. In this work, known as TD-Gammon, an ANN was used in conjunction with Temporal-Difference (TD) learning (a type of RL algorithm) to play the game of backgammon at the level of top human players.

The majority of work relating to RL, following the success of TD-Gammon, was primarily focused on theoretical results with empirical work being confined to the use of linear function approximators applied to small toy problems. However, this changed with the success of the AlexNet paper [Kri+12] in 2012, which rekindled an interest into the use of ANNs. It did not take long for RL researchers to realise the applicability of ANNs in the context of RL, and in 2013 a research group called DeepMind established the Deep-Q Network (DQN) [Mni+13], which was able to learn a wide array of Atari games from raw RGB pixel data; Something which had been impossible before the use of large non-linear function approximators. A primary reason why ANNs blossomed in the 2010s, and not in the 1990s, is due to the increase in computational power with the onset of GPUs.

It is to be stressed that ANNs do not mechanically change the nature of RL, yet rather add an extension to how RL can be performed in practice. ANNs by nature are universal function approximators by the universal approximation theorem [Hor91], which states that an ANN (with a single hidden layer) can approximate continuous functions on compact subsets of $\mathbb{R}^n$. Hence the use of ANNs in RL is to approximate some function; There are ways in which ANNs are used in the context of RL.

1. Approximate value functions: Storing the entries of a value function in a table has the limitation of only being able to store a finite number of possible values (due to memory limitations). Approximating value functions using ANNs overcomes this issue. In the DQN paper, the action-value function $Q^*(s,a)_\theta$ was approximated via an ANN.

2. Approximate the policy π_θ: As seen before, one can directly paramaterize a policy π through some function, and learn the parameters of this function via policy gradient methods; Henceforth ANNs can be used to directly represent the policy of an agent.

3 Policy Gradients

3.1 Introduction

Policy gradients is a RL paradigm for learning a policy π. A common approach for learning a policy π (as discussed in many introductory RL texts and courses) is through the approximation of value functions $V^{\pi}(s) \approx V^{w}(s)$ and $Q^{\pi}(s,a) \approx Q^{w}(s,a)$, by learning parameters w. From the construction of these value functions, a policy π can be extracted as follows $\pi^{w}(s) = \max_{a \in A} Q^{w}(s,a)$ (as discussed in the previous chapter). However, instead of learning value functions, we can directly parameterise a policy $\pi_{\theta} = P[a|s,\theta]$, for which the parameters θ can be learned via policy gradient methods. The particular appealing reasons for doing so, are the following.

1. Better convergence properties: Learning value functions can be time consuming, as they usually require complete sweeps through the state space S. An example are the generalised policy iteration methods, such as the policy iteration and value iteration algorithms briefly mentioned in section 2.3. Using policy gradient methods for learning the policy parameters is usually superior in regards to convergence.

2. Applicability to high-dimensional and continuous action spaces: A problem when learning a policy π derived from the action-value $Q^{\pi}(s,a)$, is that it requires a complete scan through the action set A, which is intractable in large discrete or continuous action spaces, due to the curse of dimensionality. [1]

Albeit the properties listed above deem to be advantageous, policy gradient methods suffer from the following issues: Convergence to a local rather than a global optimum solution, which value based approaches do not suffer from; Usually suffer from high variance in estimating updates for the parameters θ for the parameterisation π_{θ}. However, as of writing, there is a growing interest in policy gradient methods [Sch+15a; Pla+17; Sch+15b; Sch+17] due to their ability to learn relatively quickly in continuous action spaces, which is intractable with the use of value base methods.

[1] The curse of dimensionality refers to the problems difficulty increasing as a function of the number of dimensions https://en.wikipedia.org/wiki/Curse_of_dimensionality

Stationary Distribution

A stationary distribution μ of a Markov Chain remains unchanged as time progresses i.e. $\mu(x|t = i) = \mu(x|t = i + N)\ \forall\ N \geq 0$ where $i, N \in \mathbb{N}$. A policy π reduces a Markov Decision Process to a Markov Chain defining the following stationary distribution over the states $s \in S$, where $s_0 \in S$ is any initial state.

$$d^{\pi}(s) = \lim_{t \to \infty} P(s_t = s | s_0, \pi_\theta) \tag{3.1}$$

The goal in policy gradient methods, is to find a parameterised policy π_θ, with parameters θ, that maximise either of the following utility functions, dependent on the type of interaction the agent has with the environment,

$$\begin{aligned} J_1(\theta) &= V^{\pi}(s) \\ J_2(\theta) &= \sum_{s \in S} d^{\pi_\theta}(s) V^{\pi_\theta}(s) \end{aligned} \tag{3.2}$$

The utility function $J_1(\theta)$ is used in the episodic environment setting, in which the agent interacts with the environment by sampling a set $\{\tau_i\}_{i=1...N}$ of terminal experience trajectories $\tau = (s_0, a_0, r_0, s_1, a_1, r_1, \ldots, s_{T_i-1}, a_{T_i-1}, r_{T_i-1}, s_{T_i})$. The utility function $J_2(\theta)$ on the other hand, is applicable in continuing environments i.e. where $T = \infty$. In the utility function $J_2(\theta)$, $d^{\pi_\theta}(s)$ refers to the stationary distribution of the Markov Chain induced by policy π_θ as defined in definition 3.1; This can be interpreted as the probability of being in a given state $s \in S$, if the policy π_θ is applied to navigate the Markov Decision Process indefinitely.

To state the objective formally, the goal is to find parameters θ such as to maximise the expectation of the chosen utility function $J(\theta)$ (depending on a episodic or continuing environment).

$$\theta = \max_{\theta} \mathbb{E}[J(\theta)] \tag{3.3}$$

From the above equation it can be observed that the nature of the problem is an optimisation problem, henceforth learning in this context is analogous to optimisation. When dealing with optimisation problems, an ideal method for finding suitable

parameters θ is through gradient ascent (as the goal is to maximise the provided utility function).

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta) \tag{3.4}$$

Where $\nabla_\theta J(\theta) = (\frac{\partial J(\theta)}{\partial \theta_1}, \frac{\partial J(\theta)}{\partial \theta_2}, \ldots, \frac{\partial J(\theta)}{\partial \theta_n})^T$ is the column vector of partial derivates of the utility function $J(\theta)$ with respect to all parameters θ. However, the means by which this gradient $\nabla_\theta J(\theta)$ is computed, is not so obvious. An attempt that can be made, without knowing the form of $\nabla_\theta J(\theta)$ analytically, would be to apply the methods of finite differences. For each $k \in [1, n]$, the k^{th} partial derivative of the utility function $J(\theta)$ can be approximated by perturbing θ by an ϵ amount along the k^{th} dimension.

$$\frac{\partial J(\theta)}{\partial \theta_k} = \frac{J(\theta + \epsilon \hat{e}_k) - J(\theta)}{\epsilon} \tag{3.5}$$

Here the $\hat{e}_k$ is the unit basis vector corresponding to the k^{th} dimension. Although being a simple and intuitive approach and working for arbitrary (even non-differentiable) policies, this method is rather inefficient and suffers from noisy estimates. Better approaches are showcased in the following sections.

3.2 Policy Gradient

Policy Gradient Theorem

$$\begin{aligned} \nabla_\theta J(\theta) &\propto \sum_{s \in S} d^{\pi_\theta}(s) \sum_{a \in A} Q^{\pi_\theta}(s, a) \nabla_\theta \pi_\theta(a|s) \\ &= \mathbb{E}_{\pi_\theta}[Q^{\pi_\theta}(s, a) \nabla_\theta \ln \pi_\theta(a|s)] \end{aligned} \tag{3.6}$$

The problem of computing the gradient of the utility function $\nabla_\theta J(\theta)$ analytically, is the following: The policies parameters θ influence action selection and the distribution on states $d^{\pi_\theta}(s)$, both of which the utility function depend on. The effect of the parameters θ on the action selection can be computed having knowledge of the representation of the policy π_θ, however the effect of the parameters θ on the state distribution $d^{\pi_\theta}(s)$ is not straight forward, as this distribution is dependent on the dynamics of the environment, which in a model-free setting are unknown.

The policy gradient theorem [Sut+00] (defined in definition 3.6), allows us to analytically compute the gradient of the utility function $\nabla_\theta J(\theta)$, without knowledge of the effect that changes in policy parameters θ have on the state distribution $d^{\pi_\theta}(s)$. The expectation term in definition 3.6 can be derived from the first line using the following identity $\nabla_\theta \ln(w) = \frac{1}{w}\nabla_\theta w$.

$$\begin{aligned}
\nabla_\theta J(\theta) &\propto \sum_{s\in S} d^{\pi_\theta}(s) \sum_{a\in A} Q^{\pi_\theta}(s,a)\nabla_\theta \pi_\theta(a|s) \\
&= \sum_{s\in S} d^{\pi_\theta}(s) \sum_{a\in A} \pi_\theta(a|s) Q^{\pi_\theta}(s,a) \frac{\nabla_\theta \pi_\theta(a|s)}{\pi_\theta(a|s)} \\
&= \sum_{s\in S} d^{\pi_\theta}(s) \sum_{a\in A} \pi_\theta(a|s) Q^{\pi_\theta}(s,a) \nabla_\theta \ln \pi_\theta(a|s) \\
&= \mathbb{E}_\pi[Q^{\pi_\theta}(s,a)\nabla_\theta \ln \pi_\theta(a|s)]
\end{aligned} \tag{3.7}$$

$\mathbb{E}_\pi$ is a notational shorthand for $\mathbb{E}_{s\sim d^{\pi_\theta}, a\sim\pi_\theta}$, as discussed before, the actions and states sampled are both dependent on the policy π_θ. Outlined below is a proof for the policy gradient theorem. We adopt the approach of deriving the expression using $J_1(\theta) = V^\pi(s)$. The derivation using $J_2(\theta) = \sum_{s\in S} d^{\pi_\theta}(s) V^{\pi_\theta}(s)$ yields the same result, however instead of being an expression of proportionality $\nabla_\theta J(\theta) \propto \sum_{s\in S} d^{\pi_\theta}(s) \sum_{a\in A} Q^{\pi_\theta}(s,a)\nabla_\theta \pi_\theta(a|s)$ one rather arrives at an equality $\nabla_\theta J(\theta) = \sum_{s\in S} d^{\pi_\theta}(s) \sum_{a\in A} Q^{\pi_\theta}(s,a)\nabla_\theta \pi_\theta(a|s)$ [SB+98]. The reason this does not make a difference in implementation, is that any constant factor gets absorbed by the learning rate α when performing gradient ascent. For simplicity we assume no discounting $\gamma = 1$.

$$\begin{aligned}
\nabla_\theta V^\pi(s) &= \nabla_\theta \Big(\sum_{a\in A} \pi_\theta(a|s) Q^{\pi_\theta}(s,a) \Big) \\
&= \sum_{a\in A} \Big(\nabla_\theta \pi_\theta(a|s) Q^{\pi_\theta}(s,a) + \pi_\theta(a|s) \nabla_\theta Q^{\pi_\theta}(s,a) \Big) \\
&= \sum_{a\in A} \Big(\nabla_\theta \pi_\theta(a|s) Q^{\pi_\theta}(s,a) + \pi_\theta(a|s) \nabla_\theta \big(\sum_{s'\in S} p(s'|s,a)[r(s',a,s) + V^{\pi_\theta}(s')] \big) \Big) \\
&= \sum_{a\in A} \Big(\nabla_\theta \pi_\theta(a|s) Q^{\pi_\theta}(s,a) + \pi_\theta(a|s) \sum_{s'\in S} p(s'|s,a) \nabla_\theta V^{\pi_\theta}(s') \Big)
\end{aligned} \tag{3.8}$$

In the above derivation we end with a recursive formula, relating the gradient $\nabla_\theta V^\pi(s)$ to gradients $\nabla_\theta V^\pi(s')$ where s' are states succeeding state s. We unroll this

recursive formula. To reduce the amount of mathematics required, and to simplify the readability of the derivation, we make the following definition.

$$\phi(s) = \sum_{a \in A} \nabla_\theta \pi_\theta(a|s) Q^{\pi_\theta}(s,a) \tag{3.9}$$

In addition, we define $\rho^{\pi_\theta}(s \to x, k)$ as the probability of transitioning from state s to state x in k steps under policy π_θ. For $k = 0$ the probability of transitioning from state s to state x is 0 if these states are different and 1 if they are the same $\rho^{\pi_\theta}(s \to x, 0) = \delta_s^x$. When $k = 1$ we have $\rho^{\pi_\theta}(s \to x, 1) = \sum_{a \in A} \pi_\theta(a|s) p(x|s,a)$ and by the Chapman-Kolmogorov equations [2] we have that for $k = n + 1$ $\rho^{\pi_\theta}(s \to x, n+1) = \sum_{x' \in S} \rho^{\pi_\theta}(s \to x', n) \rho^{\pi_\theta}(x' \to x, 1)$.

$$\begin{aligned}
\nabla_\theta V^\pi(s) &= \sum_{a \in A} \Big(\nabla_\theta \pi_\theta(a|s) Q^{\pi_\theta}(s,a) + \pi_\theta(a|s) \sum_{s' \in S} p(s'|s,a) \nabla_\theta V^{\pi_\theta}(s') \Big) \\
&= \phi(s) + \sum_{a \in A} \pi_\theta(a|s) \sum_{s' \in S} p(s'|s,a) \nabla_\theta V^{\pi_\theta}(s') \\
&= \phi(s) + \sum_{s' \in S} \sum_{a \in A} \pi_\theta(a|s) p(s'|s,a) \nabla_\theta V^{\pi_\theta}(s') \\
&= \phi(s) + \sum_{s' \in S} \rho^{\pi_\theta}(s \to s', 1) \nabla_\theta V^{\pi_\theta}(s') \\
&= \phi(s) + \sum_{s' \in S} \rho^{\pi_\theta}(s \to s', 1) [\phi(s') + \sum_{s'' \in S} \rho^{\pi_\theta}(s' \to s'', 1) \nabla_\theta V^{\pi_\theta}(s'')] \\
&= \phi(s) + \sum_{s' \in S} \rho^{\pi_\theta}(s \to s', 1) \phi(s') + \sum_{s' \in S} \rho^{\pi_\theta}(s \to s', 1) \sum_{s'' \in S} \rho^{\pi_\theta}(s' \to s'', 1) \nabla_\theta V^{\pi_\theta}(s'') \\
&= \phi(s) + \sum_{s' \in S} \rho^{\pi_\theta}(s \to s', 1) \phi(s') + \sum_{s'' \in S} \rho^{\pi_\theta}(s \to s'', 2) \nabla_\theta V^{\pi_\theta}(s'') \\
&= \phi(s) + \sum_{s' \in S} \rho^{\pi_\theta}(s \to s', 1) \phi(s') + \sum_{s'' \in S} \rho^{\pi_\theta}(s \to s'', 2) \phi(s'') + \dots \\
&+ \sum_{s^k \in S} \rho^{\pi_\theta}(s \to s^k, k) \nabla_\theta V^{\pi_\theta}(s^k) \\
&= \sum_{x \in S} \sum_{k=0}^{\infty} \rho^{\pi_\theta}(s \to x, k) \phi(x)
\end{aligned} \tag{3.10}$$

[2] https://proofwiki.org/wiki/Chapman-Kolmogorov_Equation

Define $\eta(x) = \sum_{k=0}^{\infty} \rho^{\pi_\theta}(s \to x, k)$, which is normalised given the stationary distribution $d^{\pi_\theta}(x) = \frac{\eta(x)}{\sum_{x \in S} \eta(x)}$. Following from derivation 3.10.

$$
\begin{aligned}
\nabla_\theta V^\pi(s) &= \sum_{x \in S} \sum_{k=0}^{\infty} \rho^{\pi_\theta}(s \to x, k) \phi(x) \\
&= \sum_{x \in S} \eta(x) \phi(x) \\
&= \Big(\sum_{x \in S} \eta(x) \Big) \sum_{x \in S} d^{\pi_\theta}(x) \phi(x) \\
&\propto \sum_{x \in S} d^{\pi_\theta}(x) \phi(x) \\
&= \sum_{x \in S} d^{\pi_\theta}(x) \sum_{a \in A} \nabla_\theta \pi_\theta(a|x) Q^{\pi_\theta}(x, a)
\end{aligned}
\tag{3.11}
$$

3.3 Deterministic Policy Gradient

Deterministic Policy Gradient

$$
\begin{aligned}
\nabla_\theta J(\theta) &= \int_S \rho^\mu(s) \nabla_\theta \mu_\theta(s) \nabla_a Q^\mu(s, a)|_{a=\mu_\theta(s)} ds \\
&= \mathbb{E}_{s \sim \rho^\mu}[\nabla_\theta \mu_\theta(s) \nabla_a Q^\mu(s, a)|_{a=\mu_\theta(s)}]
\end{aligned}
\tag{3.12}
$$

The previous section established how the policy gradient $\nabla_\theta J(\theta)$ could analytically be calculated for stochastic policies $\pi_\theta(a|s)$. The deterministic policy gradient [Sil+14], outlined in definition 3.12, enables the means to compute the gradient $\nabla_\theta J(\theta)$ for deterministic policies $a = \mu_\theta(s)$; The existence of this gradient relies on the following regularity conditions: $p(s'|s, a)$, $\nabla_a p(s'|s, a)$, $\mu_\theta(s)$, $\nabla_\theta \mu_\theta(s)$, $r(s, a)$, $\nabla_a r(s, a)$ and $p_0(s)$ are continuous in all parameters. Here $p(s'|s, a)$ is the transition density function, $\mu_\theta(s)$ is a deterministic policy, $r(s, a)$ is a reward function and $p_1(s)$ is the initial density function over states $s \in S$.

The utility function $J(\theta)$ from which the gradient 3.12 is derived, is defined by the following equation.

$$\begin{aligned} J(\theta) &= \int_{\mathcal{S}} \rho^{\mu_\theta}(s) r(s, \mu_\theta(s)) ds \\ &= \mathbb{E}_{s \sim \rho^{\mu_\theta}}[\rho^{\mu_\theta}(s) r(s, \mu_\theta(s))] \end{aligned} \tag{3.13}$$

Here $\rho^{\mu}(s') = \int_{\mathcal{S}} \sum_{k=1}^{\infty} \gamma^{k-1} \rho_0(s) \rho^{\mu}(s \to s', k) ds$ is the discounted state distribution where $\rho^{\mu}(s \to s', k)$ (as in the previous section) is the probability of transitioning from state s to state s' within k steps.

A computational advantage the deterministic policy gradient has over the stochastic policy gradient, is that the deterministic policy gradient integrates over the state space, whereas the stochastic policy gradient integrates over both the state and action space. The authors of the deterministic policy gradient paper establish that the deterministic policy gradient is a limiting case of the stochastic policy gradient with the policy's variance tending to 0; In addition they derive an off-policy actor-critic algorithm that is able to outperform the stochastic policy gradient approach in high-dimensional continuous action spaces. For a proof of the deterministic policy gradient, see the appendix [3] of the paper. This proof is similar to the one outlined in the previous section.

3.4 Deep Deterministic Policy Gradients

Deep Deterministic Policy Gradients (DDPG) [Lil+15] is an actor-critic, model-free RL algorithm based on the deterministic policy gradient 3.12 outlined in the previous section. The key contribution of this paper was to combine the deterministic policy gradient with the Deep-Q Network [Mni+13], thus extending the success of the Deep-Q Network to high-dimensional continuous action spaces, as the Deep-Q Network was only applicable to discrete action spaces.

In the DDPG algorithm, both the actor and critic are represented by neural networks (as a reminder, the actor learns the policy and the critic learns the associated action-value function, which guides the learning of the policy). In order to avoid instability in training the networks, the following additions were included by the authors of the algorithm.

[3] For some strange reason the appendix is not attached to the open access paper, however it is available at the following link http://proceedings.mlr.press/v32/silver14-supp.pdf

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ.
Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer R
for episode = 1, M **do**
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state s_1
  for t = 1, T **do**
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action a_t and observe reward r_t and observe new state s_{t+1}
    Store transition (s_t, a_t, r_t, s_{t+1}) in R
    Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

    Update the target networks:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

  end for
end for

Fig. 3.1: Outline of the DDPG algorithm, extracted from the paper [Lil+15]

1. Experience replay: When training neural networks, it is assumed that the data that is fed into the network for training is independently and identically distributed (i.i.d). However, the data sampled by a policy π from an environment is not i.i.d due to the sequential nature of state transitions. The Deep-Q Network paper addressed this issue by constructing a replay buffer $\mathcal{R}$ containing sampled tuples (s_t, a_t, r_t, s_{t+1}) from which the network is trained. Sampling from this replay buffer $\mathcal{R}$ uniformly, enables the i.i.d assumption, as samples are now uncorrelated.

2. Soft target updates: As observed by the Deep-Q Network paper, directly implementing Q-Learning [WD92] with neural networks resulted in training instabilities. To overcome this issue, the authors created two copies of the neural network that learned the Q-Values, where the weights of the one were frozen for a certain period of time, before being updated with a copy of the weights of the other neural network. A similar approach was adopted by the DDPQ paper, however as this is an actor-critic model, there are a set of four neural networks: The actor network $\mu(s|\theta^\mu)$, the target actor network $\mu'(s|\theta^{\mu'})$, the critic network $Q(s, a|\theta^Q)$ and the target critic network $Q'(s, a|\theta^{Q'})$. Instead of freezing the target networks and copying the weights after a certain period of

time, a soft update approach was used, in which the target network parameters are updates as follows.

$$\begin{aligned} \theta^{Q'} &\leftarrow \tau\theta^{Q} + (1-\tau)\theta^{Q'} \\ \theta^{\mu'} &\leftarrow \tau\theta^{\mu} + (1-\tau)\theta^{\mu'} \end{aligned} \tag{3.14}$$

This update is performed after every iteration of gradient ascent on the main network's parameters with $\tau << 1$.

3. Batch normalization [IS15]: This DL technique ensures that the output of a layer within a network has a mean activation output of 0 and a standard deviation of 1. This has found to reduce training times and network instabilities in practice.

As DDPG is an off-policy method, exploration is performed by a behavioural policy, instead of the target policy; This exploration is achieved by adding noise $\mathcal{N}$ to the actor policy $\mu'(s_t) = \mu(s_t|\theta_t^{\mu}) + \mathcal{N}$. Figure 3.1 outlines the DDPQ algorithm.

4 Robotic Grasping

4.1 Introduction

The robotic grasping problem considered in this thesis is the following: A 2 degree of freedom (DOF) robotic arm with a gripper is located on a 2 dimensional surface with the objective of having to locate an object, grasp it and move it to a variable goal position. With regards to RL formalism, the environment (in its original form) has the following components in its MDP representation.

Original MDP

The Markov Decision Process describing the robotic grasping problem has the following components.

1. S: A 7 dimensional state set $S =< R_1, R_2, G, G_x, G_y, B_x, B_y >$.
2. A: A 3 dimensional action set $A =< \Delta R_1, \Delta R_2, \Delta G >$.
3. P: A deterministic transition distribution $P[S_{t+1} = s' | S_t = s, A_t = a] \in \{0, 1\}$.
4. R : A sparse reward function $r_g(s, a) = -[f_g(s) = 0]$.
5. μ: An initial state distribution $(G_x, G_y) \sim U$, $(B_x, B_y) \sim U$ and $(R_1, R_2) \sim U$.

R_1 is the rotation in radians of the lower motor (i.e. of the lower arm). R_2 is the rotation in radians of the upper motor (i.e. the upper arm). G is a binary field indicating whether the gripper is open or closed. (G_x, G_y) are the coordinates of the goal and (B_x, B_y) are the coordinates of the object. ΔR_1 is the rotation in radians applied to the lower motor (i.e. of the lower arm). ΔR_2 is the rotation in radians applied to the upper motor (i.e. the upper arm). ΔG is a binary field indicating wether to close or open the gripper. $r_g(s, a)$ is a reward function defined in a multi-goal RL setting with $g \in S$ (i.e. the goal is an element from the state set). The predicate $f_g(s) = [\|(G_x, G_y)^T - (B_x, B_y)^T\|_2 < \epsilon]$ where ϵ is some tolerance

threshold. [1] U represents the uniform distribution over the valid domain of possible values that the respective random variables can adopt.

The problem of finding a policy for the MDP outlined above turned out to be too ambitious, given the tight time constraints of this thesis and hence various adjustments were made. [2] Initially the ideas of the Hindsight Experience Replay (HER) paper [And+17] were implemented to tackle the sparse reward nature of the problem, however due to no convergence in the preliminary experiments, the sparse reward was replaced for a shaped reward [Ng+99] of the following form.

$$r(s,a,s') = f(\theta_1,\theta_2,t,h,c) + \begin{cases} \lambda_1\|l - B_l\|_2^{p_1} + \lambda_2\|\theta - B_\theta\|_2^{p_2}, & \text{if } h = 0. \\ \lambda_3\|l - G_l\|_2^{p_3}, & \text{if } h = 1. \end{cases} \tag{4.1}$$

$h \in 0, 1$ is a binary field indicating whether or not the robot has grasped the object, l is the location of the gripper, θ is the relative angle between the gripper and the object, B_l is the object location, B_θ is the absolute object angle and G_l is the goal location. $\lambda_1, \lambda_2, \lambda_3, p_1, p_2, p_3$ are tuneable hyper-parameters. f is a function taking in as arguments θ_1 and θ_2 (the coordinates in radians of the motors), t is the passed time-steps of the simulation and $c \in 0, 1$ a binary field indicating whether or not a collision between the arm and object has taken place. The objective of the function f is to enforce a continual supply of reward/punishment to ensure object completion and a balance between the two latter terms. [3]

Designing a shaped reward function is a non-trivial task and usually requires domain-specific knowledge and RL expertise. An example is the work of [Pop+17], where they engineered a cost function consisting of five complicated weighted terms to enable a robot arm to learn how to grasp and stack a brick on top of another brick. Often times reward functions lead to undesired behaviour, due to the reward function not accurately representing the task objective on hand [Amo+16a]. This phenomena was observed on multiple occasions (for different hyper-parameters values and function structure f), with some examples being.

1. The robot would swing its arm against the object to intentionally stop the simulation

[1] Here predicate refers to a logical valued function that is either true or false. In this context true implies a value of 1 and false a value 0.
[2] These adjustments were made to simplify the problem to be able to deliver a working model within the restricted time limit of the thesis.
[3] That was the idea in theory and turned out to be non-trivial to construct.

2. The robot would swing its arm around, adjusting its motors so as not to collide with the object

3. The robot would close it's gripper without performing any movement with the motors

These outcomes are a result of the robot learning that performing these undesired behaviours, it can maximise cumulative reward. This showcases the importance of constructing a shaped reward function that actually represents the objective on hand, rather than appearing to represent the objective on hand. Multiple attempts were made at designing a reward function that captures the true nature of the problem, however due to repeated failure and tight time restrictions, it was decided to rather decompose the original MDP $\mathcal{M}$ into two distinct MDPs $\mathcal{M}_1$ and $\mathcal{M}_2$, for which both valid reward functions could be established. The union of the MDPS $\mathcal{M}_1 \bigcup \mathcal{M}_2 = \mathcal{M}$ is a representation of the original MDP. [4] MDP $\mathcal{M}_1$ represents the problem of finding and grasping the object and MDP $\mathcal{M}_2$ represents the problem of moving the object to the predefined goal position. These MDPs are defined as follows.

4.1.1 Simplified Problem using MDP decomposition

MDP 1

The Markov Decision Process describing the robotic grasping problem has the following components.

1. S: A 4 dimensional state set $S =< R_1, R_2, B_x, B_y >$.
2. A: A 2 dimensional action set $A =< \Delta R_1, \Delta R_2 >$.
3. P: A deterministic transition distribution $P[S_{t+1} = s' | S_t = s, A_t = a] \in \{0, 1\}$.
4. R : A shaped reward function $r(s, a, s') = -[\alpha \|B - L\|_2 + \beta \Sigma R_i^2]$
5. μ: An initial state distribution $(B_x, B_y) \sim U*$, $R_0 = 0$ and $R_1 = \pi$.

All symbols were defined in the previous section. The reward function consists of two terms: $-\alpha\|B-L\|_2$ ensures that the L2 norm between the object and the gripper location (represented by L) gets minimised and $-\beta\Sigma R_i^2$ (adopted from the reacher gym environment [Bro+16]) is included to penalise large movements of the motors,

[4] Union in this context refers to the problem represented by the original MDP $\mathcal{M}$ being the same as the problem represented by the concatenation of MDPs $\mathcal{M}_1$ and $\mathcal{M}_2$.

ensuring smoother control. α and β are hyper-parameters ensuring proper relative scaling of the terms. [5].

The initial state distribution μ ensures that the arm always starts in the same position. The arm can grab the object if $\hat{R}_1 < r < \hat{R}_2$, where r is the distance of the object from the origin (O_x, O_y) and $\hat{R}_1$ is the lower bound and $\hat{R}_2$ is the upper bound (bounds defining boarders within the object is reachable). To simply the exploration and to ensure the defined reward function works, it was decided to spawn the object B within a subregion of the donut defined by $\hat{R}_1 < r < \hat{R}_2$. This subregion was defined by putting bounds on the angle θ (the angle the object makes in reference to the origin), which was deemed to be $\theta \in (\pi - \alpha, \pi + \alpha)$ [6]. The statement $(B_x, B_y) \sim U*$ (i.e. sampling the initial position of the object) is formalised as follows (where (O_x, O_y) are the coordinates of the origin)[7]

$$\begin{aligned} B_x &= O_x + r\cos\theta \\ B_y &= O_y - r\sin\theta \\ r &\sim U(\hat{R}_1, \hat{R}_2) \\ \theta &\sim U(\pi - \alpha, \pi + \alpha) \end{aligned} \tag{4.2}$$

To further simplify the problem at hand, it was decided to have the robot automatically close its gripper if the object is within reachable range i.e. if the predicate $[\|G - L\|_2 < \epsilon]$ (where ϵ is some tolerance threshold) evaluates to true, then the gripper closes.

[5] It was found that $\alpha = \frac{1}{150}$ and $\beta = \frac{1}{5}$ worked
[6] For the experiments $\alpha = \frac{\pi}{6}$
[7] Note the minus sign in the expression for B_y, this is due to the coordinate system used in computer graphics being different to the conventional coordinate system. This is briefly discussed in the next section.

MDP 2

The Markov Decision Process describing the robotic moving problem has the following components.

1. S: A 4 dimensional state set $S =< R_1, R_2, B_x, B_y >$.
2. A: A 2 dimensional action set $A =< \Delta R_1, \Delta R_2 >$.
3. P: A deterministic transition distribution $P[S_{t+1} = s' | S_t = s, A_t = a] \in \{0, 1\}$.
4. R : A shaped reward function $r(s, a, s') = -\left[\alpha \|B - G\|_2 + \beta \Sigma R_i^2\right]$
5. μ: An initial state distribution $(G_x, G_y) \sim U$, $(R_1, R_2) \sim U$ and $(B_x, B_y) = (L_x, L_y)$.

The MDP defined above is similar to the previous one, it contains a $4-$dimensional state space and a $2-$dimensional action space (however now the last two elements in the state space describe the location of the goal, rather than the location of the object). The transition distribution and the reward function remain the same. A difference can be observed in the initial state distribution: The goal location is sampled uniformly from the space of all possible goals $(G_x, G_y) \sim U$, as discussed before this is any location that is distance r away from the origin (O_x, O_y) with bounds $\hat{R}_1 < r < \hat{R}_2$. However, in contrast to before, we assume the goal location (G_x, G_y) to take on any possible value in the space of all reachable coordinates and hence $\theta \in [0, 2\pi)$. This is formalised as follows.

$$\begin{aligned} G_x &= O_x + r\cos\theta \\ G_y &= O_y - r\sin\theta \\ r &\sim U(\hat{R}_1, \hat{R}_2) \\ \theta &\sim U(0, 2\pi) \end{aligned} \tag{4.3}$$

In this MDP, it is given that the arm can be in any initial configuration and hence $(R_1, R_2) \sim U$ which is formalised as.

$$\begin{aligned} R_1 &\sim U(0, 2\pi) \\ R_2 &\sim U(0, 2\pi) \end{aligned} \tag{4.4}$$

In addition, in this MDP the object is attached to the robot (i.e the robot is assumed to already have grasped the object).

$$\begin{aligned} B_x &= L_x \\ B_y &= B_y \end{aligned} \tag{4.5}$$

These equations continue to be satisfied throughout time-steps $t > 0$ (and not just for $t = 0$).

The problem that MDP $\mathcal{M}_2$ is describing is more alluring than that of MPD $\mathcal{M}_1$, as the robot has to learn how to move the object from any starting position to any ending position within the vicinity of legal coordinates, instead of starting in a fixed position and having to learn how to move to a location in a subregion of legal coordinates.

4.2 Simulation

4.2.1 Introduction

Simulations provide imitations of a physical phenomenon in virtual environments. Simulations play a vital part in RL, as they provide the primary mechanism in which RL algorithms are trained and tested. The most established simulation environments (in the context of RL) are the ones developed by OpenAI, known as the gym [Bro+16]. The gym is a collection of open-source game environments that share a common interface, enabling the development of general RL algorithms. Two fundamental contributions that OpenAI gym bring forward, are the following.

1. Delivering a diverse set of environments that are intuitive to use. This is analogous to big datasets (such as the imageNet [Den+09]) in supervised learning, that have driven large progress.

2. Providing standardisation for comparing RL algorithms in the literature. Slight changes in reward functions, action spaces or environment dynamics can dramatically alter the difficulty of a given task, thus making it difficult to compare various RL algorithms if the environments are not consistent.

In the context of RL, simulations can be used in conjunction with the physical world. The agent would first be trained in the simulated environment, before being placed in the physical environment for deployment or further training. In this instance, the simulated environment would be modelled after the physical environment. There are two difficulties in doing so.

1. Expertise: Being given a physical environment, one has to handcraft a replica in a virtual setting. Such crafting is usually non-trivial as it requires domain specific knowledge. Before training any RL model, one has to create these environments opposed to using something like gym, which makes this a tedious

process. A potential downfall, is that the replicated virtual environment might not capture the true nature of the physical world and hence becomes a bad model for training. This brings us to the next point.

2. Expense: In order to develop good models of the physical world, many times research groups resort to using expensive physics engines that capture the dynamics of the physical world. A popular example is the Multi-Joint dynamics with Contact (MuJoCo) physics engine [Tod+12] which has been used in a variety of RL related research projects [Lil+15; Dua+16; Lev+16; Sch+15b]. Being a powerful tool, the price for the engine ranges between 500 to 12000 dollars (depending on the license type), which is a steep financial burden for anyone who wants to perform RL research related to robotics.

A custom simulator was built for the robotic grasp problem (outlined in the previous section), that tackles the two problems mentioned above: The simulator is modest, in that it does not require an extensive array of expertise. It is low cost, as it was built using an open-source game engine without the addition of complicated dynamics (hence not requiring a physics engine). This showcases that RL robotics research can be performed without the need of expensive proprietary software.

The simulator consists of three main components: The environment (which is fundamental to any RL simulator), the arm and the object which both form part of the environment. The following subsections outline some of the construction that was used in the development of the virtual robotic arm. All development was carried out in a modest open-source python game engine called PyGame [8], with the source code attached in the appendix.

On a closing note: The coordinate system used in computer graphics differs from the standard 2D coordinate system that one usually works with. The conventional 2D coordinate system is the right-handed Cartesian coordinate system, where the x-axis goes to the right and the y-axis goes to the top. In computer graphics however, the x-axis goes to the right and the y-axis goes to the bottom. The following subsections take this into account, hence the importance of mentioning this. The changes with such a coordinate system is that the origin is now the top left corner; increasing the y coordinate of an object will translate the object downwards; and the counter clockwise rotation $R(\theta)$ through angle θ about the origin now performs a clockwise rotation.

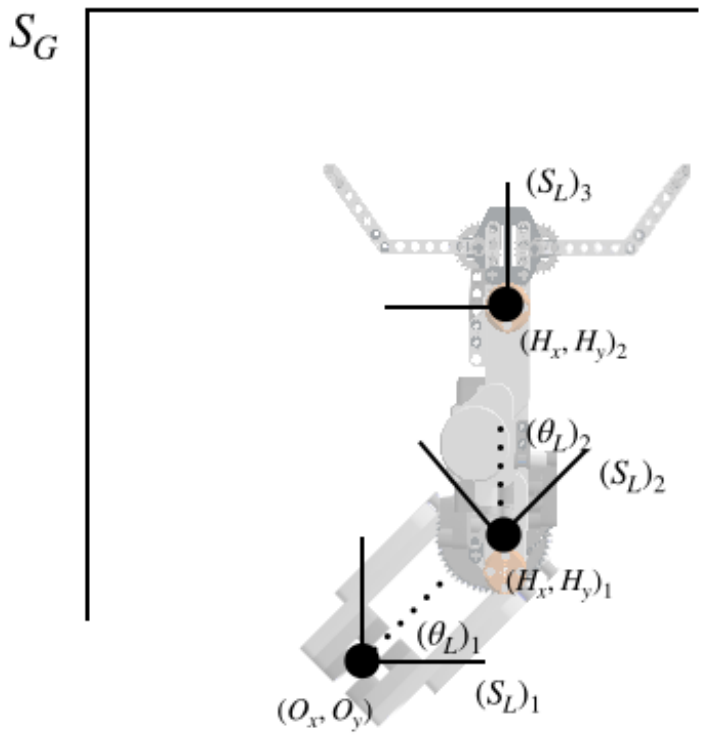

Fig. 4.1: Overview of the robotic arm, with all the local coordinate systems $(S_L)_i$, hinge points $(H_x, H_y)_i$ and elevation angles $(\theta_L)_i$.

4.2.2 The Arm

The arm is comprised of four main components: The lower arm, the upper arm and the gripper, where the gripper is further decomposed into a left and right gripper part. All components live in a global coordinate system (S_G), with each having their own local coordinate system $(S_L)_i$. The global coordinate system (S_G) is the main coordinate system, in that each object is placed within this coordinate system for rendering purposes; However it also enables vital functionality such as checking for collision and rotating objects. The local coordinate systems $(S_L)_i$ provide a mechanism of bookkeeping for each individual part, that takes care of simplifying the calculations involving rotations. Each arm object has a rotation $(\theta_L)_i$ described in the local coordinate system $(S_L)_i$ and a hinge point $(H_x, H_y)_i$ described in the global coordinate system (S_G).

[8]https://en.wikipedia.org/wiki/Pygame

The lower arm lives in a local coordinate system $(S_L)_1$ with its origin centred at location (O_x, O_y) in the global coordinate system (S_G). The upper arm lives in the local coordinate system $(S_L)_2$ which is rotated by $(\theta_L)_1$ and centred at $(H_x, H_y)_1$. The gripper (encapsulating both the left and right gripper) lives in a local coordinate system $(S_L)_3$ that is rotated by $(\theta_L)_1 + (\theta_L)_2$ and centred at $(H_x, H_y)_2$. This is visually portrayed in figure 4.1.

All the arm components are derived from the same class [9], which provides common shared functionality to all components. Each arm object is characterised by 8 variables.

1. Coordinates (x, y) which are in reference to the global coordinate system (S_G). These points are dynamically calculated after each update to the arm.

2. An offset O_f which is a length describing how much to translate the object by, as to move it to the correct hinge point.

3. A scale S which is the length of the arm object

4. A global angle A_g which characterises the rotation of the arm object in the global coordinate system (S_G).

5. A local source angle A_s which characterises the current rotation of the arm object in the local coordinate system $(S_L)_i$.

6. A local destination angle A_d which characterises the destination rotation (i.e the goal angle to rotate to) of the arm object in the local coordinate system $(S_L)_i$.

7. A rate of change $d\theta$ encoding how quickly the arm object is to rotate.

The main purpose of the class is to enable a mechanism for rotating and translating the arm objects. Whilst the source angle A_s is not equal to the destination angle A_d, the source angle is updated as follows $A_s \leftarrow A_s + d\theta$ with rotation transformation 4.6 being applied [10].

$$R(\theta) = \begin{bmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix} \tag{4.6}$$

[9] In Object Orientated Programming (OOP) a class provides a blueprint for an object. The object is said to be instantiated from a class, if a copy is made.

[10] This transformation is performed in the global coordinate system (S_G)

After each rotation, new coordinates (x, y) are computed for each arm part. The new coordinates for the lower arm are computed as follows.

$$\begin{aligned} x &= O_x + cos(A_{g1})O_{f1} \\ y &= O_y - sin(A_{g1})O_{f1} \end{aligned} \tag{4.7}$$

The updated coordinates for the upper arm are calculated as follows.

$$\begin{aligned} x &= O_x + cos(A_{g2})O_{f2} + S_1 cos(A_{g1}) \\ y &= O_y - sin(A_{g2})O_{f2} - S_1 sin(A_{g1}) \end{aligned} \tag{4.8}$$

The updated coordinates for the gripper parts are calculated as follows.

$$\begin{aligned} x &= O_x + cos(A_{g3})O_{f3} + S_1 cos(A_{g1}) + S_2 cos(A_{g2}) \\ y &= O_y - sin(A_{g3})O_{f3} - S_1 sin(A_{g1}) - S_2 sin(A_{g2}) \end{aligned} \tag{4.9}$$

4.2.3 The Gripper

As mentioned before, the gripper is comprised of two dynamic parts, the left and right gripper part. The same rotations and translations as before are applied, however an additional translation scheme is introduced to move each gripper part apart from each other and to the desired location on the upper arm. If this additional translation scheme would not be applied, both gripper parts would be fixed at the hinge point of the upper arm. We introduce a horizontal offset Δx and a vertical offset Δy that translates the left gripper part Δx units to the left and the right gripper part Δx units to the right and both parts Δy upwards (in regards to the local coordinate system in which the gripper parts are situated). See figure 4.2 for a visual depiction

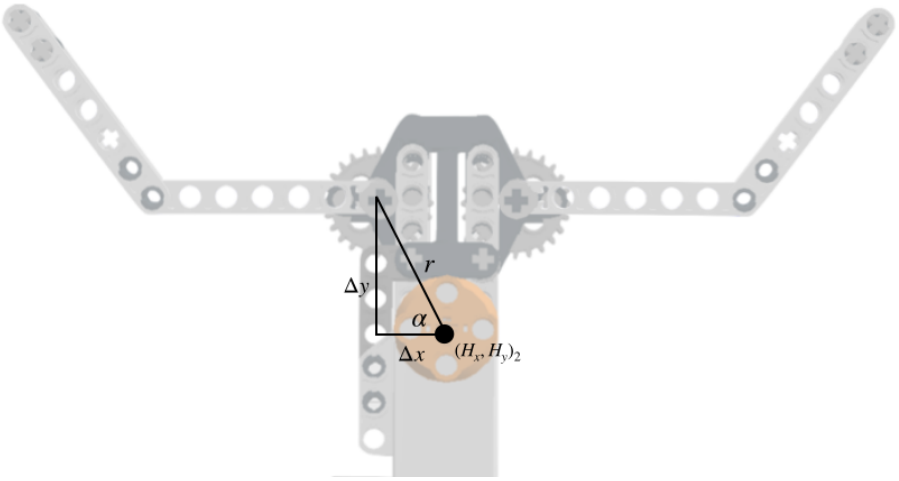

Fig. 4.2: Overview of the robotic gripper, portraying how the left gripper is translated from the hinge point of the upper arm. A similar translation is applied for the right gripper part.

of this translation process. These offsets are appended to the coordinates of the left gripper part (x_L, y_L) and right gripper part (x_R, y_R) as follows.

$$
\begin{aligned}
x_L &\leftarrow cos(A_{s1} + A_{s2} + \alpha)r \\
y_L &\leftarrow -sin(A_{s1} + A_{s2} + \alpha)r \\
x_R &\leftarrow cos(A_{s1} + A_{s2} - \alpha)r \\
y_R &\leftarrow -sin(A_{s1} + A_{s2} - \alpha)r \\
\alpha &= tan^{-1}(\frac{\Delta y}{\Delta x}) \\
r &= \sqrt{(\Delta x)^2 + (\Delta y)^2}
\end{aligned}
\tag{4.10}
$$

The main functionality of the gripper is to grasp the object. In the initial construction of the simulator, it was attempted to include laws of physics to give realistic contact collision between the gripper (including the arm) and the object. This turned out to be a non-trivial task as one would have to take into account (to name a few): The

friction of the surface, the angular momentum of the object and material properties of the robot. After much research (reading about physics, discussing with colleagues, reaching out to physics stack exchange and designing code), it was decided to adopt a simpler approach to checking whether or not the object had been grasped; After all, every model is wrong (including the more complicated one with realistic physics) and some are useful.

In order to check if the object is in a valid grasping position, a circular region of radius T_1 (referred to as the intra tolerance) displaced by an offset of T_2 (referred to as the inter tolerance) from the top of the upper arm, was constructed. If the object falls within this circle, it is considered graspable. The circle centre coordinates (x_E, y_E) are calculated as follows.

$$\begin{aligned} x_E &= x_2 + (\frac{S_2}{2} + T_2 + r_o)cos(A_{s1} + A_{s2}) \\ y_E &= y_2 - (\frac{S_2}{2} + T_2 + r_o)sin(A_{s1} + A_{s2}) \end{aligned} \tag{4.11}$$

In these equations, (x_2, y_2) are the coordinates of the upper arm, S_2 is the length of the upper arm, r_o is the radius of the object and A_{s1} and A_{s2} are the source angles of the lower and upper arm respectively. If the robot is issued a command to close the gripper and the following predicate holds true

$$[\sqrt{(x_B - x_E)^2 + (y_B - y_E)^2} < T_1] \tag{4.12}$$

then the object is considered to be grasped and is attached to the upper arm (here (x_B, y_B) are the coordinates of the object).

4.3 Physical Development

A robotic arm with a grasping hand was constructed out of various materials. The robot itself was constructed from a robotics kit called the Mindtorms NXT [11]. The robot arm is situated on top of a wooden board, that provides support and forms the region for the object and goals. As per the problem description 4.1, this robot is a 2 DOF arm: It has one motor controlling the lower arm and another motor that rotates the upper arm. The third motor is connected to a worm drive that is able

[11] https://en.wikipedia.org/wiki/Lego _Mindstorms

to open and close the grasping hand. The motors used are all servo motors, which allow for precise control of angular position via sensory position feedback; Knowing the rotation of the motors is a requirement described in the problem definition of the MDPs. The motors are controlled via a computer known as the NXT. The NXT is a modest 32-bit micro controller with 64Kb of RAM and 256KB of flash memory. The NXT receives which commands to issue to the motors from a main computer running the simulation models, via a USB connection. The communication between the NXT and the main computer was established via the control library nxt-python [12] coupled with the communication library pyusb [13].

Adjustments to the simulator described in the previous section had to be made, such that the robot and simulator coincide: Firstly, the rotations in the simulator (which are in radians) had to be mapped to the rotations applied to the physical motors (which are in degrees).

$$\theta_D = \alpha \frac{\theta_R}{2\pi} G_R \mathcal{I}_D \tag{4.13}$$

θ_D is the output rotation in degrees and θ_R is the input rotation in radians. $G_R = 7$ is the gear ratio: The number of turns the driven gear makes relative to the driver gear. $\mathcal{I}_D = 360$ is one rotation in degrees. Here $\alpha = 0.99$ is a hyper-parameter, which was found via trial and error. This parameter takes into account various factors of the physical environment, which would otherwise hinder the mapping from the virtual to the physical space from being exact. Secondly, the dimensions of the arm parts used in simulation had to relatively align with the scale dimensions of the arm parts of the physical robot. In order to achieve this, the physical robot was rebuilt in a Computer-Aided Design (CAD) software known as LDD [14]. Lastly, the grasp circle (outlined in subsection 4.2.3) was deemed to have a radius of $T_1 = 20$mm and an offset of $T_2 = 15$mm.

4.4 DDPG Implementation

The DDPG implementation was tacking from a Github repository [15] and modified to the problem on hand. All implementation was done using the PyTorch [16] library. The neural networks used for the actor and critic were composed of three layers. Data normalisation was applied to avoid numeric instabilities during training. Neural

[12] https://github.com/eelviny/nxt-python
[13] https://github.com/pyusb/pyusb
[14] https://en.wikipedia.org/wiki/Lego _Digital _Designer
[15] https://github.com/vy007vikas/PyTorch-ActorCriticRL
[16] https://pytorch.org

weights were initialised from a uniform distribution. For training the Adam optimizer [KB14] was used with a learning rate of $\eta = 0.001$ and a mini-batch size of 128. The following hyper parameters were used as outlined in the DDPG algorithm 3.1: $\tau = 0.001$ and $\gamma = 0.99$.

To ensure adequate exploration the following scheme was adopted: With 0.3 the same exploration as adopted as in the DDPG paper was performed, sampling correlated noise from the Ornstein-Uhlenbeck process [UO30] and adding it to the target policy. With the remaining 0.7 probability an action was uniformly sampled from the hypercube of possible actions; This ensured adequate exploration.

4.5 Experiments and Results

This section outlines the experiments conducted, in benchmarking the effectiveness of training policies π_1 and π_2 (respectively associated with MDPs $\mathcal{M}_1$ and $\mathcal{M}_2$ described in subsection 4.1.1) in simulation and applying it to the physical robot.

4.5.1 Training

All training was conducted on a server consisting of 2vCPUs with 4Gb of memory. This server was hosted on Digital Ocean [17] and accessed remotely using the terminal (for issuing commands) and Cyberduck [18] (for file transfer).

A main training and testing class was created. This class could delegate the task of training the policies π_i for each MDP $\mathcal{M}_i$ and combine these policies to act in the overall MDP $\mathcal{M}$ (i.e. the original problem description). For both policies, training was conducted using 10000 episodes, where each episode consisted of 30 time steps.

As performed in RL literature, the total cumulative reward per episode (analogous to training accuracy in supervised learning) was recorded over the various training episodes. The total reward over episodes is portrayed in figure 4.3 for both policies π_1 and π_2. It can be observed, that the policies learn relatively quickly, with both reaching a total reward score of approximately -10 after 500 episodes. [19]

[17] https://en.wikipedia.org/wiki/DigitalOcean
[18] https://en.wikipedia.org/wiki/Cyberduck
[19] The figure was clipped to have a domain of 2000 episodes, rather than 10000, as results did not vary over the extended interval.

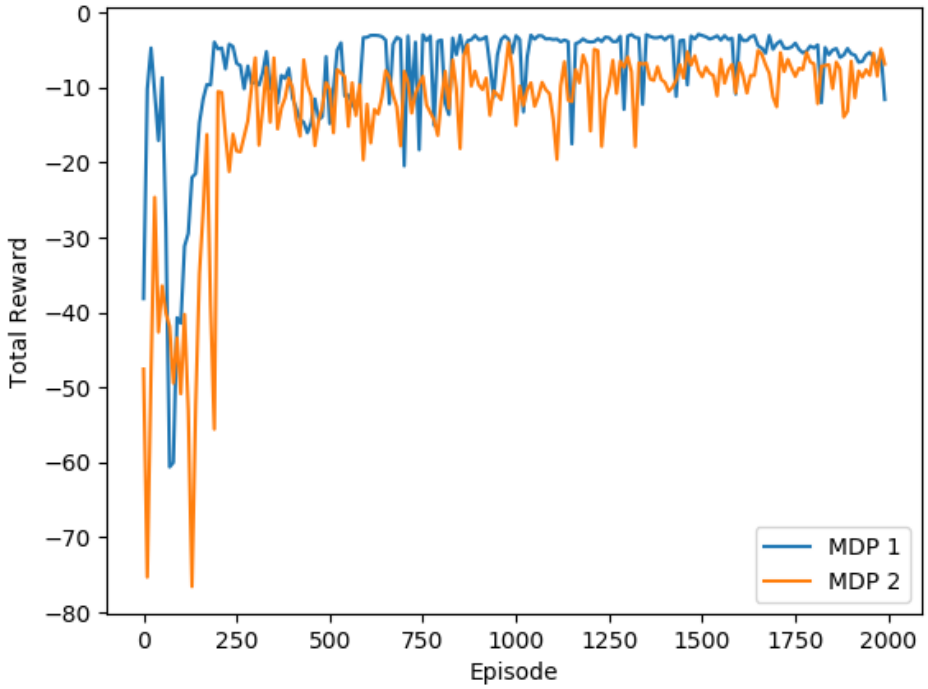

Fig. 4.3: Training results of of policies for the MDPs described in section 4.1.1

4.5.2 Testing

To validate the effectiveness of the policies learned, they were first benchmarked in the simulator. For testing purposes 1000 trials were run. In each trial the object was instantiated with a random location $(B_x, B_y) \sim U^*$ with the arm joints set to default positions $R_1 = 0$ and $R_2 = \pi$. The goal coordinates were also set to a random coordinate $(G_x, G_y) \sim U$. See subsection 4.1.1 for an explanation as to how these coordinates are generated. In these preliminary experiments, 185 of the episodes failed i.e. the robot was not able to grab the object and move it to the goal location; All the other attempts were successful. To define the goodness of success, the displacement of the final object position in reference to the attempted goal position was measured. For the successful attempts, the mean displacement was 14.73px with a standard deviation of 10.04px. This showcases that in simulation, the robot is able to get the object to the goal position with good precision (given that the reachable radius is 280px), however this precision is rather variable.

For the experiments in the physical environment, the object coordinate was fixed to be $(O_x - 180\text{px}, O_y)$ for every trial. The main reason for this, is that it simplified the testing procedure. As physical tests are cumbersome to run, a small sample of 7 trials was conducted. Physical tests were cumbersome to run, primarily because the physical environment had to manually be reset for every trial, in contrast to the

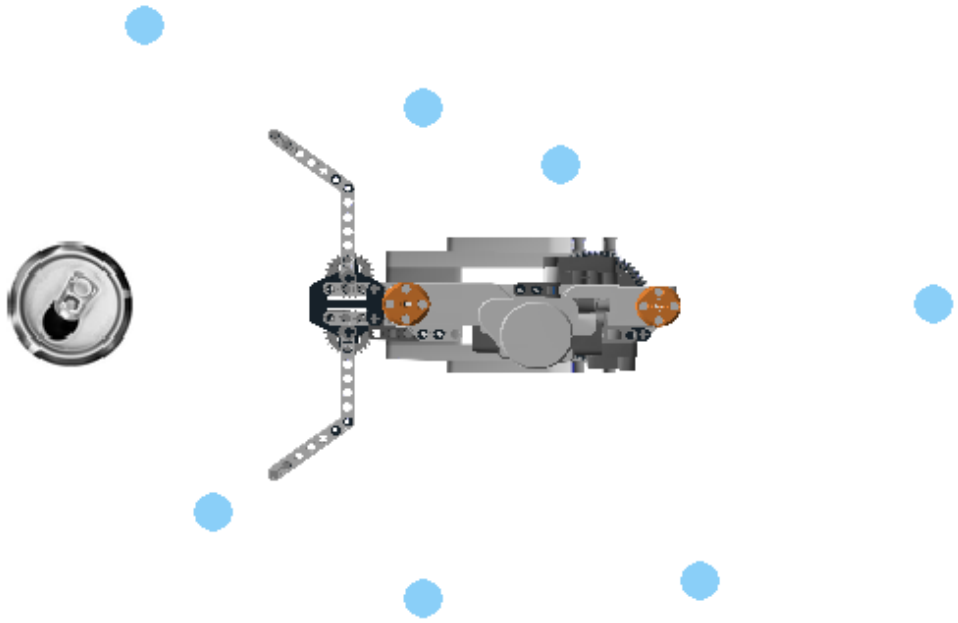

Fig. 4.4: The goal coordinates that were used in the physical experiments.

virtual environment which could automatically reset itself. The goal coordinates (G_x, G_y) were manually chosen in a way, as to cover most of the legal coordinate space. See figure x that portrays where all these points coordinates are located, in reference to the robot. These coordinates, alongside the displacements of the end object position to the goal position in the physical and virtual setting are recorded in table 4.1. In order to check for correlation between the physical and virtual displacements, a scatter plot was generated, which is portrayed in figure 4.5. It can be observed that there is a linear relationship between the physical and virtual displacements. This indicates that the error made in the physical environment is due to the same reason as in the virtual environment, and not that the virtual model constructed misrepresents the problem on hand. This showcases that with modest expertise and no expense, a simulation for the robotic grasping problem can be constructed, that encompasses all learning, with the learned policies being applicable in the physical environment (without additional fine tuning).

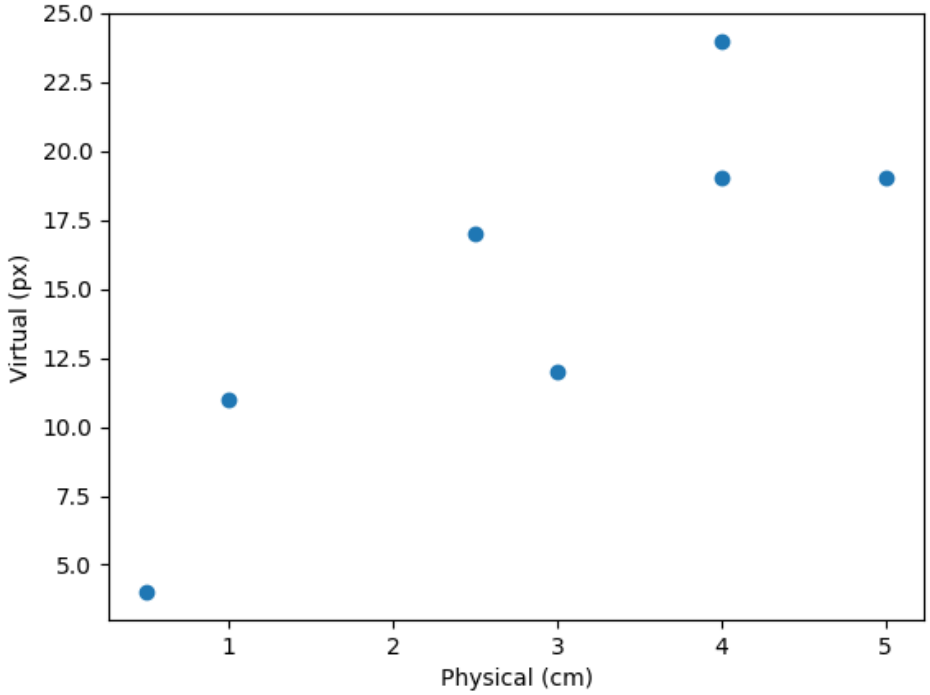

Fig. 4.5: Scatter plot of the virtual displacements against the physical displacements, where the displacement is the difference between the final object position and the goal position.

goal coordinates	physical offset (cm)	virtual offset (px)
(258, 258)	0.5	4
(400, 300)	5	19
(470, 329)	2.5	17
(660, 400)	4	19
(541, 541)	4	24
(400, 550)	3	12
(293, 506)	1	11

Tab. 4.1: Attempted goal coordinates alongside recorded offsets

5 Conclusion

In this work the problem of applying Deep Reinforcement Learning to the problem of robotic grasping in physical environments was tackled through the use of a self built simulator, avoiding proprietary expensive simulation and physics engines. The experiments performed showcase that it is possible to learn meaningful behaviour in a self built simulator. This motivates the idea that RL applied to robotics can be performed without the need of expensive software.

The robotic grasping problem was initially attempted using a sparse binary reward, by implementing the Hindsight Experience Replay (HER) algorithm [And+17], without any success. Reasons for no convergence could have been due to the neural networks not being deep enough, errors in the implementation of the algorithm or a lack of computational power. Moving to a shaped reward setting proved difficult, as a function encoding the objective on hand could not be found. Decomposing the robot grasp problem into two sub problems solved the problem: First learn a policy that is able to find the object, then learn another policy that is able to move the object to the desired goal location.

Future work would focus on developing a generic simulator that can be customised to personal projects: Defining the robot's abilities and dimensions, inputing the reward function to be used and setting certain physical properties that should be incorporated in simulation.

Bibliography

[Amo+16a] Dario Amodei, Chris Olah, Jacob Steinhardt, et al. „Concrete problems in AI safety“. In: *arXiv preprint arXiv:1606.06565* (2016) (cit. on p. 30).

[Amo+16b] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, et al. „Deep speech 2: End-to-end speech recognition in english and mandarin“. In: *International Conference on Machine Learning*. 2016, pp. 173–182 (cit. on p. 11).

[And+17] Marcin Andrychowicz, Filip Wolski, Alex Ray, et al. „Hindsight experience replay“. In: *Advances in Neural Information Processing Systems*. 2017, pp. 5048–5058 (cit. on pp. 30, 47).

[Bah+14] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. „Neural machine translation by jointly learning to align and translate“. In: *arXiv preprint arXiv:1409.0473* (2014) (cit. on p. 11).

[Bro+16] Greg Brockman, Vicki Cheung, Ludwig Pettersson, et al. „Openai gym“. In: *arXiv preprint arXiv:1606.01540* (2016) (cit. on pp. 31, 34).

[Den+09] Jia Deng, Wei Dong, Richard Socher, et al. „Imagenet: A large-scale hierarchical image database“. In: *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*. Ieee. 2009, pp. 248–255 (cit. on pp. 12, 34).

[Dua+16] Yan Duan, Xi Chen, Rein Houthooft, John Schulman, and Pieter Abbeel. „Benchmarking deep reinforcement learning for continuous control“. In: *International Conference on Machine Learning*. 2016, pp. 1329–1338 (cit. on pp. 3, 35).

[Gro13] Stephen Grossberg. „Recurrent neural networks“. In: *Scholarpedia* 8.2 (2013), p. 1888 (cit. on p. 12).

[He+16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. „Deep residual learning for image recognition“. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778 (cit. on p. 12).

[HM95] Jun Han and Claudio Moraga. „The influence of the sigmoid function parameters on the speed of backpropagation learning“. In: *International Workshop on Artificial Neural Networks*. Springer. 1995, pp. 195–201 (cit. on p. 13).

[Hoc+01] Sepp Hochreiter, Yoshua Bengio, Paolo Frasconi, Jürgen Schmidhuber, et al. *Gradient flow in recurrent nets: the difficulty of learning long-term dependencies*. 2001 (cit. on p. 13).

[Hor91] Kurt Hornik. „Approximation capabilities of multilayer feedforward networks“. In: *Neural networks* 4.2 (1991), pp. 251–257 (cit. on p. 17).

[IS15] Sergey Ioffe and Christian Szegedy. „Batch normalization: Accelerating deep network training by reducing internal covariate shift“. In: *arXiv preprint arXiv:1502.03167* (2015) (cit. on p. 27).

[Kak02] Sham M Kakade. „A natural policy gradient“. In: *Advances in neural information processing systems*. 2002, pp. 1531–1538 (cit. on p. 7).

[KB14] Diederik P Kingma and Jimmy Ba. „Adam: A method for stochastic optimization“. In: *arXiv preprint arXiv:1412.6980* (2014) (cit. on p. 42).

[Kri+12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. „Imagenet classification with deep convolutional neural networks“. In: *Advances in neural information processing systems*. 2012, pp. 1097–1105 (cit. on pp. 12, 13, 17).

[LeC+98] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. „Gradient-based learning applied to document recognition“. In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324 (cit. on p. 12).

[Lev+16] Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. „End-to-end training of deep visuomotor policies“. In: *The Journal of Machine Learning Research* 17.1 (2016), pp. 1334–1373 (cit. on pp. 3, 35).

[Lil+15] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, et al. „Continuous control with deep reinforcement learning“. In: *arXiv preprint arXiv:1509.02971* (2015) (cit. on pp. 3, 8, 25, 26, 35).

[Mni+13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, et al. „Playing atari with deep reinforcement learning“. In: *arXiv preprint arXiv:1312.5602* (2013) (cit. on pp. 17, 25).

[Ng+99] Andrew Y Ng, Daishi Harada, and Stuart Russell. „Policy invariance under reward transformations: Theory and application to reward shaping“. In: *ICML*. Vol. 99. 1999, pp. 278–287 (cit. on p. 30).

[Pla+17] Matthias Plappert, Rein Houthooft, Prafulla Dhariwal, et al. „Parameter space noise for exploration“. In: *arXiv preprint arXiv:1706.01905* (2017) (cit. on p. 19).

[Pop+17] Ivaylo Popov, Nicolas Heess, Timothy Lillicrap, et al. „Data-efficient deep reinforcement learning for dexterous manipulation“. In: *arXiv preprint arXiv:1704.03073* (2017) (cit. on p. 30).

[Ren+15] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. „Faster r-cnn: Towards real-time object detection with region proposal networks“. In: *Advances in neural information processing systems*. 2015, pp. 91–99 (cit. on p. 12).

[SB+98] Richard S Sutton, Andrew G Barto, et al. *Reinforcement learning: An introduction*. MIT press, 1998 (cit. on pp. 2, 8, 11, 22).

[Sch+15a] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. „High-dimensional continuous control using generalized advantage estimation“. In: *arXiv preprint arXiv:1506.02438* (2015) (cit. on p. 19).

[Sch+15b] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. „Trust region policy optimization“. In: *International Conference on Machine Learning*. 2015, pp. 1889–1897 (cit. on pp. 3, 7, 19, 35).

[Sch+17] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. „Proximal policy optimization algorithms“. In: *arXiv preprint arXiv:1707.06347* (2017) (cit. on p. 19).

[Sil+14] David Silver, Guy Lever, Nicolas Heess, et al. „Deterministic policy gradient algorithms“. In: *ICML*. 2014 (cit. on p. 24).

[Sil+16] David Silver, Aja Huang, Chris J Maddison, et al. „Mastering the game of Go with deep neural networks and tree search“. In: *nature* 529.7587 (2016), p. 484 (cit. on p. 1).

[SL06] István Szita and András Lörincz. „Learning Tetris using the noisy cross-entropy method“. In: *Neural computation* 18.12 (2006), pp. 2936–2941 (cit. on p. 7).

[Sut+00] Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. „Policy gradient methods for reinforcement learning with function approximation“. In: *Advances in neural information processing systems*. 2000, pp. 1057–1063 (cit. on pp. 7, 22).

[Sze+15] Christian Szegedy, Wei Liu, Yangqing Jia, et al. „Going deeper with convolutions“. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, pp. 1–9 (cit. on p. 12).

[Tes95] Gerald Tesauro. „Temporal difference learning and TD-Gammon“. In: *Communications of the ACM* 38.3 (1995), pp. 58–68 (cit. on p. 17).

[Tod+12] Emanuel Todorov, Tom Erez, and Yuval Tassa. „Mujoco: A physics engine for model-based control“. In: *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on*. IEEE. 2012, pp. 5026–5033 (cit. on pp. 2, 35).

[UO30] George E Uhlenbeck and Leonard S Ornstein. „On the theory of the Brownian motion“. In: *Physical review* 36.5 (1930), p. 823 (cit. on p. 42).

[WD92] Christopher JCH Watkins and Peter Dayan. „Q-learning“. In: *Machine learning* 8.3-4 (1992), pp. 279–292 (cit. on pp. 8, 26).

[Wie+08] Daan Wierstra, Tom Schaul, Jan Peters, and Juergen Schmidhuber. „Natural evolution strategies“. In: *Evolutionary Computation, 2008. CEC 2008.(IEEE World Congress on Computational Intelligence). IEEE Congress on*. IEEE. 2008, pp. 3381–3387 (cit. on p. 7).

[Wil92] Ronald J Williams. „Simple statistical gradient-following algorithms for connectionist reinforcement learning“. In: *Machine learning* 8.3-4 (1992), pp. 229–256 (cit. on p. 7).

[WP09] Kevin Wampler and Zoran Popović. „Optimal gait and form for animal locomotion“. In: *ACM Transactions on Graphics (TOG)*. Vol. 28. 3. ACM. 2009, p. 60 (cit. on p. 7).

6 Appendix

6.1 Gym

6.1.1 arm.py

```
import numpy as np
import pygame

class Arm():

    ROTATION_SPEED = None
    __ROTATION_SPEED_TRAINING = 0.4
    __ROTATION_SPEED_RENDER = 0.04

    __LOWER_ARM_W = 135
    __LOWER_ARM_H = 70
    __UPPER_ARM_W = 190
    __UPPER_ARM_H = 68

    def __init__(self, x, y, default_arm_position, attach_beer, render =
        True):

        if(render):
            Arm.ROTATION_SPEED = Arm.__ROTATION_SPEED_RENDER
        else:
            Arm.ROTATION_SPEED = Arm.__ROTATION_SPEED_TRAINING

        if(default_arm_position):
            lower_angle = 0
            upper_angle = np.pi
        else:
            lower_angle = np.random.uniform(low = 0, high = 2 * np.pi)
            upper_angle = np.random.uniform(low = 0, high = 2 * np.pi)

        self.lower_arm = ArmPart('gym/images/lower_arm.png', x, y,
            Arm.__LOWER_ARM_W, Arm.__LOWER_ARM_H, angle = lower_angle, off
            = 20)
```

```
        self.upper_arm = ArmPart('gym/images/upper_arm.png', x, y,
            Arm.__UPPER_ARM_W, Arm.__UPPER_ARM_H, reference_parts =
            [self.lower_arm], angle = upper_angle, off = 36)
        self.gripper = Gripper(x, y, [self.lower_arm, self.upper_arm],
            attach_beer)

        self.arm_group = pygame.sprite.Group()
        self.gripper_group = pygame.sprite.Group()

        self.arm_group.add(self.lower_arm)
        self.arm_group.add(self.upper_arm)

        self.gripper_group.add(self.gripper.left_gripper)
        self.gripper_group.add(self.gripper.right_gripper)

        self.lower_arm.rotate(0)
        self.update()

    def is_stationary(self):
        # print("Stat: " + str(self.lower_arm.is_stationary() and
            self.upper_arm.is_stationary() and
            self.gripper.is_stationary()))
        return self.lower_arm.is_stationary() and
            self.upper_arm.is_stationary() and self.gripper.is_stationary()

    def is_collide(self, beer):

        # Ignore any sort of collisions if the arm has grabbed the beer
        if(self.gripper.has_beer == True):
            return False

        # Check if the lower or upper arm have collided with the object
        if(len(pygame.sprite.spritecollide(beer, self.arm_group, False,
            pygame.sprite.collide_mask)) > 0):
            return True

        # Check if the gripper has collided with the beer (and the beer is
            not attached)
        if(len(pygame.sprite.spritecollide(beer, self.gripper_group, False,
            pygame.sprite.collide_mask)) > 0):
            return True

        return False

    def rotate_lower_arm(self, radians, dtheta = None):
        if(dtheta == None):
            dtheta = Arm.ROTATION_SPEED
```

```
            if(radians > 0):
                self.lower_arm.dtheta = dtheta
            else:
                self.lower_arm.dtheta = - dtheta
            self.lower_arm.dst_angle += radians

        def rotate_upper_arm(self, radians, dtheta = None):
            if(dtheta == None):
                dtheta = Arm.ROTATION_SPEED
            if(radians > 0):
                self.upper_arm.dtheta = dtheta
            else:
                self.upper_arm.dtheta = - dtheta
            self.upper_arm.dst_angle += radians

        def close(self, beer):
            self.gripper.close(beer)

        def update(self, beer = None):
            # Update simulations associated with the individual arm parts
            self.arm_group.update()
            self.gripper_group.update()
            # Update top level parts to rotate with lower level parts (as this
                happens in the real physical system)
            self.upper_arm.rotate(self.lower_arm.src_angle)
            self.gripper.update(self.lower_arm.src_angle +
                self.upper_arm.src_angle, beer)

        def render(self, surface):
            self.arm_group.draw(surface)
            self.gripper_group.draw(surface)

    class ArmPart(pygame.sprite.Sprite):
        """
        A class for storing relevant arm segment information.
        """
        def __init__(self, image, x, y, w, h, reference_parts = None, angle =
            0, off = 0):
            super().__init__()
            self.base_image = pygame.image.load(image)
            self.base_image = pygame.transform.scale(self.base_image, (w, h))
            self.base_image.set_alpha(128)
            self.image = self.base_image
            self.rect = self.image.get_rect()
            self.x = x
            self.y = y
            self.off = off
```

```
        self.length = self.image.get_rect()[2]
        self.scale = self.length - 2 * off
        self.offset = self.scale / 2.0
        #self.offset = self.length / 2 - offset

        self.reference_parts = reference_parts

        self.src_angle = angle # The current relative angle of the arm part
        self.dst_angle = angle # The final relative angle of the arm part
        self.vis_angle = angle # The current absolute angle of the arm part
        self.dtheta = 0 # The speed at which the arm part rotates
        self.recenter()

    def is_stationary(self):
        return self.src_angle == self.dst_angle

    def recenter(self):
        # Base reference
        self.rect.center = (self.x, self.y)

        # Relative self adjustment
        self.rect.center += np.array([np.cos(self.vis_angle) * self.offset,
            -np.sin(self.vis_angle) * self.offset])

        # Adjustments based on other reference points
        if(self.reference_parts is not None):
            for arm_part in self.reference_parts:
                self.rect.center += np.array([arm_part.scale *
                    np.cos(arm_part.vis_angle), -arm_part.scale *
                    np.sin(arm_part.vis_angle)])

    def update(self):
        if(self.src_angle != self.dst_angle):
            self.src_angle += self.dtheta
            self.rotate()

        if(self.dtheta < 0 and self.src_angle <= self.dst_angle):
            self.dtheta = 0
            self.src_angle = self.dst_angle
        elif(self.dtheta > 0 and self.src_angle >= self.dst_angle):
            self.dtheta = 0
            self.src_angle = self.dst_angle

    def rotate(self, *args):
        vis_angle = self.src_angle
```

```
        if(len(args) == 1):
            vis_angle += args[0]

        self.image = pygame.transform.rotozoom(self.base_image,
            np.degrees(vis_angle), 1)
        self.rect = self.image.get_rect()
        self.vis_angle = vis_angle
        self.recenter()

class Gripper:

    INTRA_TOLERANCE = 20 # Radius of the epislon ball
    INTER_TOLERANCE = 15 # Displacement of the epsilon ball from the
        gripper surface

    GRIPPER_HOR_GAP = 16
    GRIPPER_VER_GAP = 10

    INIT_GRIPPER_ANGLE = 0.5 * np.pi
    CLOSED_GRIPPER_ANGLE = 0.2 * np.pi

    __GRIPPER_W = 80
    __GRIPPER_H = 45 + 37

    def __init__(self, x, y, reference_parts, attach_beer):
        self.left_gripper = ArmPart('gym/images/left_gripper.png', x, y,
            Gripper.__GRIPPER_W, Gripper.__GRIPPER_H, reference_parts =
            reference_parts)
        self.right_gripper = ArmPart('gym/images/right_gripper.png', x, y,
            Gripper.__GRIPPER_W, Gripper.__GRIPPER_H, reference_parts =
            reference_parts)

        self.alpha = np.arctan(Gripper.GRIPPER_VER_GAP /
            Gripper.GRIPPER_HOR_GAP)
        self.radius = np.sqrt(Gripper.GRIPPER_HOR_GAP ** 2 +
            Gripper.GRIPPER_VER_GAP ** 2)

        if(attach_beer):
            self.left_gripper.src_angle = Gripper.CLOSED_GRIPPER_ANGLE
            self.left_gripper.dst_angle = Gripper.CLOSED_GRIPPER_ANGLE
            self.right_gripper.src_angle = - Gripper.CLOSED_GRIPPER_ANGLE
            self.right_gripper.dst_angle = - Gripper.CLOSED_GRIPPER_ANGLE
            self.closed = True
            self.has_beer = True
        else:
            self.left_gripper.src_angle = Gripper.INIT_GRIPPER_ANGLE
            self.left_gripper.dst_angle = Gripper.INIT_GRIPPER_ANGLE
```

```
        self.right_gripper.src_angle = - Gripper.INIT_GRIPPER_ANGLE
        self.right_gripper.dst_angle = - Gripper.INIT_GRIPPER_ANGLE
        self.closed = False
        self.has_beer = False

def is_stationary(self):
    return self.left_gripper.is_stationary() and
        self.right_gripper.is_stationary()

def recenter(self, angle):

    # Relative self adjustment
    self.left_gripper.rect.center += np.array([np.cos(angle +
        self.alpha) * self.radius, -np.sin(angle + self.alpha) *
        self.radius])
    self.right_gripper.rect.center += np.array([np.cos(angle -
        self.alpha) * self.radius, -np.sin(angle - self.alpha) *
        self.radius])

def update(self, angle, beer = None):
    self.left_gripper.rotate(angle)
    self.right_gripper.rotate(angle)
    self.recenter(angle)

    if(beer is not None and self.has_beer):
        x_b, y_b = self.get_beer_coordinates(beer, 0)
        beer.set_angle(angle)
        beer.set_pos(x_b, y_b)

def get_beer_coordinates(self, beer, displacement):
    r = beer.radius
    w = self.left_gripper.reference_parts[-1].length / 2
    h = w + displacement + r
    ang = self.left_gripper.reference_parts[0].src_angle +
        self.left_gripper.reference_parts[1].src_angle
    x_b = self.left_gripper.reference_parts[-1].rect.centerx + h *
        np.cos(ang)
    y_b = self.left_gripper.reference_parts[-1].rect.centery - h *
        np.sin(ang)

    return (x_b, y_b)

def beer_in_reach(self, beer):
    x, y = beer.get_pos()
    x_b, y_b = self.get_beer_coordinates(beer, Gripper.INTER_TOLERANCE)
    return ((x - x_b) ** 2 + (y - y_b) ** 2 < Gripper.INTRA_TOLERANCE
        ** 2)
```

```
    def close(self, beer):
        if(self.closed):
            return
        self.closed = True
        self.left_gripper.dtheta = - Arm.ROTATION_SPEED
        self.left_gripper.dst_angle = Gripper.CLOSED_GRIPPER_ANGLE
        self.right_gripper.dtheta = Arm.ROTATION_SPEED
        self.right_gripper.dst_angle = - Gripper.CLOSED_GRIPPER_ANGLE

        # Attach beer if it is within a given region centered at the tip of
            the upper arm
        if(self.beer_in_reach(beer)):
            self.has_beer = True
```

6.1.2 beer.py

```
import pygame
import numpy as np
from arm import Gripper

class Beer(pygame.sprite.Sprite):

    __BEER_RADIUS = 32

    ALPHA = np.pi / 6 # Spawning angle
    LOWER_GEN_BOUND = 142 # Spawning lower bound for radius

    def __init__(self, x, y, arm):
        super().__init__()
        self.base_image = pygame.image.load('gym/images/beer.png')
        self.base_image = pygame.transform.scale(self.base_image, (2 *
            Beer.__BEER_RADIUS, 2 * Beer.__BEER_RADIUS))
        self.image = self.base_image
        self.rect = self.image.get_rect()
        self.radius = self.rect.w / 2
        self.__init(x, y, arm) # Set the initial position
        while(arm.is_collide(self)): # Keep initialising until there is no
            collision
            self.__init(x, y, arm)

    def __init(self, x, y, arm):
        init_r = np.random.uniform(low = Beer.LOWER_GEN_BOUND, high =
            arm.lower_arm.scale + arm.upper_arm.scale)
        init_theta = np.random.uniform(low = np.pi - Beer.ALPHA, high =
            np.pi + Beer.ALPHA)
```

```
        init_x = int(x + init_r * np.cos(init_theta))
        init_y = int(y - init_r * np.sin(init_theta))
        self.set_pos(init_x, init_y)
        self.theta = init_theta

    def set_pos(self, x, y):
        self.rect.centerx = x
        self.rect.centery = y

    def set_angle(self, angle):
        self.image = pygame.transform.rotozoom(self.base_image,
            np.degrees(angle), 1)
        self.rect = self.image.get_rect()

    def get_pos(self):
        return (self.rect.centerx, self.rect.centery)

    def update(self):
        pass

    def render(self, surface):
        surface.blit(self.image, self.rect)
```

6.1.3 environment.py

```
import pygame
import pygame.locals
import sys
import numpy as np

from arm import Arm, Gripper
from beer import Beer

import robot.motor_control as robot

class Environment():

    RENDER_WIDTH = 800
    RENDER_HEIGHT = 800

    ACTION_SCALES = np.float32([0.3 * np.pi, 0.3 * np.pi])
    STATE_SCALES = np.float32([2 * np.pi, 2 * np.pi, RENDER_WIDTH,
        RENDER_HEIGHT])

    REWARD_SCALE = 0.7
```

```
# Goals used for the physical experiments
GOAL_1 = (int(RENDER_WIDTH / 2 - 200 * np.cos(np.pi / 4)),
    int(RENDER_HEIGHT / 2 - 200 * np.sin(np.pi / 4)))
GOAL_2 = (int(RENDER_WIDTH / 2), int(RENDER_HEIGHT / 2 - 100))
GOAL_3 = (int(RENDER_WIDTH / 2 + 100 * np.cos(np.pi / 4)),
    int(RENDER_HEIGHT / 2 - 100 * np.sin(np.pi / 4)))
GOAL_4 = (int(RENDER_WIDTH / 2 + 260), int(RENDER_HEIGHT / 2))
GOAL_5 = (int(RENDER_WIDTH / 2 + 200 * np.cos(np.pi / 4)),
    int(RENDER_HEIGHT / 2 + 200 * np.sin(np.pi / 4)))
GOAL_6 = (int(RENDER_WIDTH / 2), int(RENDER_HEIGHT / 2 + 150))
GOAL_7 = (int(RENDER_WIDTH / 2 - 150 * np.cos(np.pi / 4)),
    int(RENDER_HEIGHT / 2 + 150 * np.sin(np.pi / 4)))

def __init__(self, config = 1, robot_control = False, render = True):
    print("Goal 1: " + str(Environment.GOAL_1))
    print("Goal 2: " + str(Environment.GOAL_2))
    print("Goal 3: " + str(Environment.GOAL_3))
    print("Goal 4: " + str(Environment.GOAL_4))
    print("Goal 5: " + str(Environment.GOAL_5))
    print("Goal 6: " + str(Environment.GOAL_6))
    print("Goal 7: " + str(Environment.GOAL_7))

    self.config = config
    self.robot_control = robot_control
    if(self.robot_control):
        robot.connect()
    self.render = render
    pygame.init()
    if(render):
        self.surface =
            pygame.display.set_mode((Environment.RENDER_WIDTH,
            Environment.RENDER_HEIGHT))
    self.fpsClock = pygame.time.Clock()
    self.reset()

def reset(self, beer_location = None, goal_location = None):
    if(self.config == 0 or self.config == 1):
        default_arm_position = True
        attach_beer = False
    elif(self.config == 2):
        default_arm_position = False
        attach_beer = True
    self.arm = Arm(Environment.RENDER_WIDTH / 2,
        Environment.RENDER_HEIGHT / 2, default_arm_position,
        attach_beer, render = self.render)
    self.beer = Beer(Environment.RENDER_WIDTH / 2,
        Environment.RENDER_HEIGHT / 2, self.arm)
```

```
        self.goal = self.__sample_goal(Environment.RENDER_WIDTH / 2,
            Environment.RENDER_HEIGHT / 2, self.arm)

        if(beer_location is not None):
            self.beer.set_pos(beer_location[0], beer_location[1])

        if(goal_location is not None):
            self.goal = goal_location

        return self.__get_observation()

    def sample_action(self):
        return np.float32([np.random.uniform(low = -1, high = 1) for _ in
            range(len(Environment.ACTION_SCALES))])

    def step(self, action, render_goal = True, render_goals = False,
        render_grasp_circle = False, render_boarders = False):
        action *= Environment.ACTION_SCALES
        rot_lower = action[0]
        rot_upper = action[1]

        # Automatically close the gripper if the beer is in reach
        if(self.arm.gripper.has_beer == False and
            self.arm.gripper.beer_in_reach(self.beer)):
            self.arm.close(self.beer)
            if(self.robot_control):
                robot.close_gripper()
        else:
            if(np.linalg.norm(np.float32(self.goal) -
                np.float32(self.beer.get_pos())) > 20):
                self.arm.rotate_lower_arm(rot_lower)
                self.arm.rotate_upper_arm(rot_upper)
                if(self.robot_control):
                    robot.rotate_lower_arm(rot_lower)
                    robot.rotate_upper_arm(rot_upper)

        collided = self.arm.is_collide(self.beer)

        if(self.arm.is_stationary() or collided):
            if(self.render):
                self._render(render_goal, render_goals,
                    render_grasp_circle, render_boarders)

        # Update until action is completed
        while(self.arm.is_stationary() == False and collided == False):
            self.arm.update(beer = self.beer)
            collided = self.arm.is_collide(self.beer)
```

```
            if(self.render):
                self._render(render_goal, render_goals,
                    render_grasp_circle, render_boarders)

        observation = self.__get_observation()

        if(self.config == 0):
            if(self.arm.gripper.has_beer == False):
                target_location =
                    np.float32(self.arm.gripper.get_beer_coordinates(self.beer,
                    Gripper.INTER_TOLERANCE))
            else:
                target_location = self.goal
        elif(self.config == 1):
            target_location =
                np.float32(self.arm.gripper.get_beer_coordinates(self.beer,
                Gripper.INTER_TOLERANCE))
        elif(self.config == 2):
            target_location = self.goal

        reward = Environment.REWARD_SCALE *
            self.__get_reward(self.beer.get_pos(), target_location,
            rot_lower, rot_upper, collided)

        return (observation, reward, collided)

    def __get_observation(self):

        if(self.config == 0):
            if(self.arm.gripper.has_beer == False):
                obs = (self.arm.lower_arm.src_angle % (2 * np.pi),
                    self.arm.upper_arm.src_angle % (2 * np.pi),
                    self.beer.rect.centerx, self.beer.rect.centery)
            else:
                obs = (self.arm.lower_arm.src_angle % (2 * np.pi),
                    self.arm.upper_arm.src_angle % (2 * np.pi),
                    self.goal[0], self.goal[1])
        elif(self.config == 1):
            obs = (self.arm.lower_arm.src_angle % (2 * np.pi),
                self.arm.upper_arm.src_angle % (2 * np.pi),
                self.beer.rect.centerx, self.beer.rect.centery)
        elif(self.config == 2):
            obs = (self.arm.lower_arm.src_angle % (2 * np.pi),
                self.arm.upper_arm.src_angle % (2 * np.pi), self.goal[0],
                self.goal[1])

        return np.float32(obs)
```

```
def __get_reward(self, beer_location, target_location, r1, r2,
    collided):
    beer_location = np.float32(beer_location)
    target_location = np.float32(target_location)
    reward_dist = - np.linalg.norm(beer_location - target_location) /
        120
    reward_ctrl = - np.square(np.float32((r1, r2))).sum() / 5

    return reward_dist + reward_ctrl

def __sample_goal(self, x, y, arm):
    init_r = np.random.uniform(low = Beer.LOWER_GEN_BOUND, high =
        arm.lower_arm.scale + arm.upper_arm.scale)
    init_theta = np.random.uniform(low = 0, high = 2 * np.pi)
    init_x = int(x + init_r * np.cos(init_theta))
    init_y = int(y - init_r * np.sin(init_theta))
    return (init_x, init_y)

def _render(self, render_goal, render_goals, render_grasp_circle,
    render_boarders):
    white = (255, 255, 255)
    self.surface.fill(white)

    if(render_goal):
        self.__render_goal()

    if(render_goals):
        self.__render_goals()

    if(render_grasp_circle):
        self.__render_grasp_circle()

    if(render_boarders):
        self.__render_boarders()

    self.beer.render(self.surface)
    self.arm.render(self.surface)

    # Check for quit + bug fix
    for event in pygame.event.get():
        if event.type == pygame.locals.QUIT:
            pygame.quit()
            sys.exit()

    pygame.display.update()
    self.fpsClock.tick(60)
```

```
    def __render_goal(self):
        blue = (135, 206, 250)
        pygame.draw.circle(self.surface, blue, self.goal, 20)

    def __render_grasp_circle(self):
        purple = (219, 112, 147)
        pygame.draw.circle(self.surface, purple,
            np.int32(self.arm.gripper.get_beer_coordinates(self.beer,
            Gripper.INTER_TOLERANCE)), Gripper.INTRA_TOLERANCE)

    def __render_boarders(self):
        red = (250, 128, 114)
        white = (255, 255, 255)
        pygame.draw.circle(self.surface, red, (int(Environment.RENDER_WIDTH
            / 2), int(Environment.RENDER_HEIGHT / 2)),
            self.arm.lower_arm.scale + self.arm.upper_arm.scale)
        pygame.draw.circle(self.surface, white,
            (int(Environment.RENDER_WIDTH / 2),
            int(Environment.RENDER_HEIGHT / 2)), Beer.LOWER_GEN_BOUND)

    def __render_goals(self):
        blue = (135, 206, 250)
        pygame.draw.circle(self.surface, blue, Environment.GOAL_1, 10)
        pygame.draw.circle(self.surface, blue, Environment.GOAL_2, 10)
        pygame.draw.circle(self.surface, blue, Environment.GOAL_3, 10)
        pygame.draw.circle(self.surface, blue, Environment.GOAL_4, 10)
        pygame.draw.circle(self.surface, blue, Environment.GOAL_5, 10)
        pygame.draw.circle(self.surface, blue, Environment.GOAL_6, 10)
        pygame.draw.circle(self.surface, blue, Environment.GOAL_7, 10)
```

6.1.4 motor_control.py

```
import numpy as np
import nxt.locator
from nxt.motor import *

MOTOR_POWER = 40
LOWER_ARM_PORT = PORT_A
UPPER_ARM_PORT = PORT_C
GRIPPER_PORT = PORT_B

lower_motor = None
upper_motor = None
gripper_motor = None
```

```
def connect():
    global lower_motor, upper_motor, gripper_motor
    brick = nxt.locator.find_one_brick()
    lower_motor = Motor(brick, LOWER_ARM_PORT)
    upper_motor = Motor(brick, UPPER_ARM_PORT)
    gripper_motor = Motor(brick, GRIPPER_PORT)

def rotate_lower_arm(radians):
    degrees = (radians / (2 * np.pi)) * (7 * 360) * 0.99
    if(degrees > 0):
        lower_motor.turn(MOTOR_POWER, np.abs(degrees), brake = False)
    else:
        lower_motor.turn(-MOTOR_POWER, np.abs(degrees), brake = False)

def rotate_upper_arm(radians):
    degrees = (radians / (2 * np.pi)) * (7 * 360) * 1
    if(degrees > 0):
        upper_motor.turn(MOTOR_POWER, np.abs(degrees), brake = False)
    else:
        upper_motor.turn(-MOTOR_POWER, np.abs(degrees), brake = False)

def close_gripper():
    gripper_motor.turn(MOTOR_POWER, int(4.3 * 360))
```

6.2 DDPG

6.2.1 buffer.py

```
   import numpy as np
import random
from collections import deque


class MemoryBuffer:

   MAX_BUFFER = 100000

   def __init__(self, size = MAX_BUFFER):
      self.buffer = deque(maxlen=size)
      self.maxSize = size
      self.len = 0

   def sample(self, count):
      batch = []
```

```
        count = min(count, self.len)
        batch = random.sample(self.buffer, count)
        s_arr = np.float32([arr[0] for arr in batch])
        a_arr = np.float32([arr[1] for arr in batch])
        r_arr = np.float32([arr[2] for arr in batch])
        s1_arr = np.float32([arr[3] for arr in batch])
        return s_arr, a_arr, r_arr, s1_arr

    def len(self):
        return self.len

    def add(self, s, a, r, s1):
        transition = (s,a,r,s1)
        self.len += 1
        if self.len > self.maxSize:
            self.len = self.maxSize
        self.buffer.append(transition)
```

6.2.2 main.py

```
    import numpy as np
import train
import buffer
import sys
sys.path.insert(0, './gym')
from environment import Environment
import time

class TestPolicies():

    EPISODES = 1
    TIME_STEPS = 60

    def __init__(self, pol1, pol2):
        self.env = Environment(0, robot_control = True, render = True)
        self.trainer1 = train.Trainer(1, len(Environment.STATE_SCALES),
            len(Environment.ACTION_SCALES), None)
        self.trainer1.load_models(pol1)
        self.trainer2 = train.Trainer(2, len(Environment.STATE_SCALES),
            len(Environment.ACTION_SCALES), None)
        self.trainer2.load_models(pol2)

    def test(self):

        fails = 0
        distances = []
```

```
for episode in range(TestPolicies.EPISODES):

   print("EPISODE: ", episode)

   # For the physical experiments
   beer_location = (int(Environment.RENDER_WIDTH / 2 - 180),
       int(Environment.RENDER_HEIGHT / 2))
   goal_location = Environment.GOAL_2

   # For the virtual experiments
   # beer_location = None
   # goal_location = None

   state = self.env.reset(beer_location = beer_location,
       goal_location = goal_location)

   failed = False
   for t in range(TestPolicies.TIME_STEPS):

      state /= self.env.STATE_SCALES

      if(self.env.arm.gripper.has_beer == False):
         action = self.trainer1.get_exploitation_action(state)
      else:
         action = self.trainer2.get_exploitation_action(state)

      new_state, reward, done = self.env.step(action)
      state = new_state

      if(done):
         failed = True
         fails += 1
         break

      if(self.env.robot_control):
         time.sleep(0.5)

   if(failed == False):
      d = np.linalg.norm(np.float32(self.env.goal) -
          np.float32(self.env.beer.get_pos()))
      distances.append(d)

print("FAILED: " + str(fails))
print("Mean: " + str(np.mean(distances)))
print("STD: " + str(np.std(distances)))
```

```
class TrainPolicy():

   EPISODES = 10000
   TIME_STEPS = 30
   ALPHA = 0.3

   def __init__(self, config):
      self.config = config
      self.env = Environment(self.config, render = False)
      self.ram = buffer.MemoryBuffer()
      self.trainer = train.Trainer(config, len(Environment.STATE_SCALES),
          len(Environment.ACTION_SCALES), self.ram)

   def train(self, alpha = ALPHA):
      for episode in range(TrainPolicy.EPISODES):

         print("EPISODE: ", episode)

         tot_reward = 0
         state = self.env.reset() / self.env.STATE_SCALES

         for t in range(TrainPolicy.TIME_STEPS):

            if(episode % 10 == 0):
               action = self.trainer.get_exploitation_action(state)
            else:
               if(np.random.uniform() <= alpha):
                  action = self.trainer.get_exploration_action(state)
               else:
                  action = self.env.sample_action()

            new_state, reward, _ = self.env.step(action)
            new_state /= Environment.STATE_SCALES
            tot_reward += reward

            self.ram.add(state, action, reward, new_state)
            state = new_state
            self.trainer.optimize()

         # Log results
         if(episode % 10 == 0):
            with open("results_" + str(self.config) + ".txt", 'a+') as f:
               f.write(str(episode) + ", " + str(tot_reward) + "\n")

         # Save models
         if(episode % 100 == 0):
            self.trainer.save_models(episode)
```

```
        print("Total Reward: " + str(tot_reward))

# trainer1 = TrainPolicy(1)
# trainer1.train()

# trainer2 = TrainPolicy(2)
# trainer2.train()

# for i in range(20):
#  time.sleep(1)
#  print(i)

tester = TestPolicies(5000, 9900)
tester.test()
```

6.2.3 model.py

```
   import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np

EPS = 0.003

def fanin_init(size, fanin=None):
   fanin = fanin or size[0]
   v = 1. / np.sqrt(fanin)
   return torch.Tensor(size).uniform_(-v, v)

class Critic(nn.Module):

   def __init__(self, state_dim, action_dim):
      super(Critic, self).__init__()

      self.state_dim = state_dim
      self.action_dim = action_dim

      self.fcs1 = nn.Linear(state_dim,256)
      self.fcs1.weight.data = fanin_init(self.fcs1.weight.data.size())
      self.fcs2 = nn.Linear(256,128)
      self.fcs2.weight.data = fanin_init(self.fcs2.weight.data.size())

      self.fca1 = nn.Linear(action_dim,128)
      self.fca1.weight.data = fanin_init(self.fca1.weight.data.size())
```

```
        self.fc2 = nn.Linear(256,128)
        self.fc2.weight.data = fanin_init(self.fc2.weight.data.size())

        self.fc3 = nn.Linear(128,1)
        self.fc3.weight.data.uniform_(-EPS,EPS)

    def forward(self, state, action):
        s1 = F.relu(self.fcs1(state))
        s2 = F.relu(self.fcs2(s1))
        a1 = F.relu(self.fca1(action))
        x = torch.cat((s2,a1),dim=1)
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x


class Actor(nn.Module):

    def __init__(self, state_dim, action_dim):
        super(Actor, self).__init__()

        self.state_dim = state_dim
        self.action_dim = action_dim

        self.fc1 = nn.Linear(state_dim,256)
        self.fc1.weight.data = fanin_init(self.fc1.weight.data.size())

        self.fc2 = nn.Linear(256,128)
        self.fc2.weight.data = fanin_init(self.fc2.weight.data.size())

        self.fc3 = nn.Linear(128,64)
        self.fc3.weight.data = fanin_init(self.fc3.weight.data.size())

        self.fc4 = nn.Linear(64,action_dim)
        self.fc4.weight.data.uniform_(-EPS,EPS)

    def forward(self, state):
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        x = F.relu(self.fc3(x))
        action = F.tanh(self.fc4(x))

        return action
```

6.2.4 train.py

```
   import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.autograd import Variable

import numpy as np
import math

import utils
import model

BATCH_SIZE = 128
LEARNING_RATE = 0.001
GAMMA = 0.99
TAU = 0.001

class Trainer:

   def __init__(self, config, state_dim, action_dim, ram):
      self.config = config
      self.ram = ram
      self.noise = utils.OrnsteinUhlenbeckActionNoise(action_dim)

      self.actor = model.Actor(state_dim, action_dim)
      self.target_actor = model.Actor(state_dim, action_dim)
      self.actor_optimizer =
          torch.optim.Adam(self.actor.parameters(),LEARNING_RATE)

      self.critic = model.Critic(state_dim, action_dim)
      self.target_critic = model.Critic(state_dim, action_dim)
      self.critic_optimizer =
          torch.optim.Adam(self.critic.parameters(),LEARNING_RATE)

      utils.hard_update(self.target_actor, self.actor)
      utils.hard_update(self.target_critic, self.critic)

   def get_exploitation_action(self, state):
      state = Variable(torch.from_numpy(state))
      action = self.target_actor.forward(state).detach()
      return action.data.numpy()

   def get_exploration_action(self, state):
      state = Variable(torch.from_numpy(state))
      action = self.actor.forward(state).detach()
      new_action = action.data.numpy() + self.noise.sample()
      return new_action
```

```
    def optimize(self):
        s1, a1, r1, s2 = self.ram.sample(BATCH_SIZE)
        s1 = Variable(torch.from_numpy(s1))
        a1 = Variable(torch.from_numpy(a1))
        r1 = Variable(torch.from_numpy(r1))
        s2 = Variable(torch.from_numpy(s2))

        a2 = self.target_actor.forward(s2).detach()
        next_val = torch.squeeze(self.target_critic.forward(s2, a2).detach())
        y_expected = r1 + GAMMA*next_val
        y_predicted = torch.squeeze(self.critic.forward(s1, a1))
        loss_critic = F.smooth_l1_loss(y_predicted, y_expected)
        self.critic_optimizer.zero_grad()
        loss_critic.backward()
        self.critic_optimizer.step()

        pred_a1 = self.actor.forward(s1)
        loss_actor = -1 * torch.sum(self.critic.forward(s1, pred_a1))
        self.actor_optimizer.zero_grad()
        loss_actor.backward()
        self.actor_optimizer.step()

        utils.soft_update(self.target_actor, self.actor, TAU)
        utils.soft_update(self.target_critic, self.critic, TAU)

    def save_models(self, episode_count):
        torch.save(self.target_actor.state_dict(), './Models_' +
            str(self.config) + '/' + str(episode_count) + '_actor.pt')
        torch.save(self.target_critic.state_dict(), './Models_' +
            str(self.config) + '/' + str(episode_count) + '_critic.pt')

    def load_models(self, episode):
        self.actor.load_state_dict(torch.load('./Models_' + str(self.config)
            + '/' + str(episode) + '_actor.pt'))
        self.critic.load_state_dict(torch.load('./Models_' +
            str(self.config) + '/' + str(episode) + '_critic.pt'))
        utils.hard_update(self.target_actor, self.actor)
        utils.hard_update(self.target_critic, self.critic)
```

6.2.5 utils.py

```
    import numpy as np
import torch
import shutil
import torch.autograd as Variable
```

```
def soft_update(target, source, tau):
    for target_param, param in zip(target.parameters(),
        source.parameters()):
        target_param.data.copy_(target_param.data * (1.0 - tau) + param.data
            * tau)


def hard_update(target, source):
    for target_param, param in zip(target.parameters(),
        source.parameters()):
          target_param.data.copy_(param.data)

# Based on
    http://math.stackexchange.com/questions/1287634/implementing-ornstein-uhlenbeck-in-matlab
class OrnsteinUhlenbeckActionNoise:

    def __init__(self, action_dim, mu = 0, theta = 0.15, sigma = 0.2):
        self.action_dim = action_dim
        self.mu = mu
        self.theta = theta
        self.sigma = sigma
        self.X = np.ones(self.action_dim) * self.mu

    def reset(self):
        self.X = np.ones(self.action_dim) * self.mu

    def sample(self):
        dx = self.theta * (self.mu - self.X)
        dx = dx + self.sigma * np.random.randn(len(self.X))
        self.X = self.X + dx
        return self.X
```