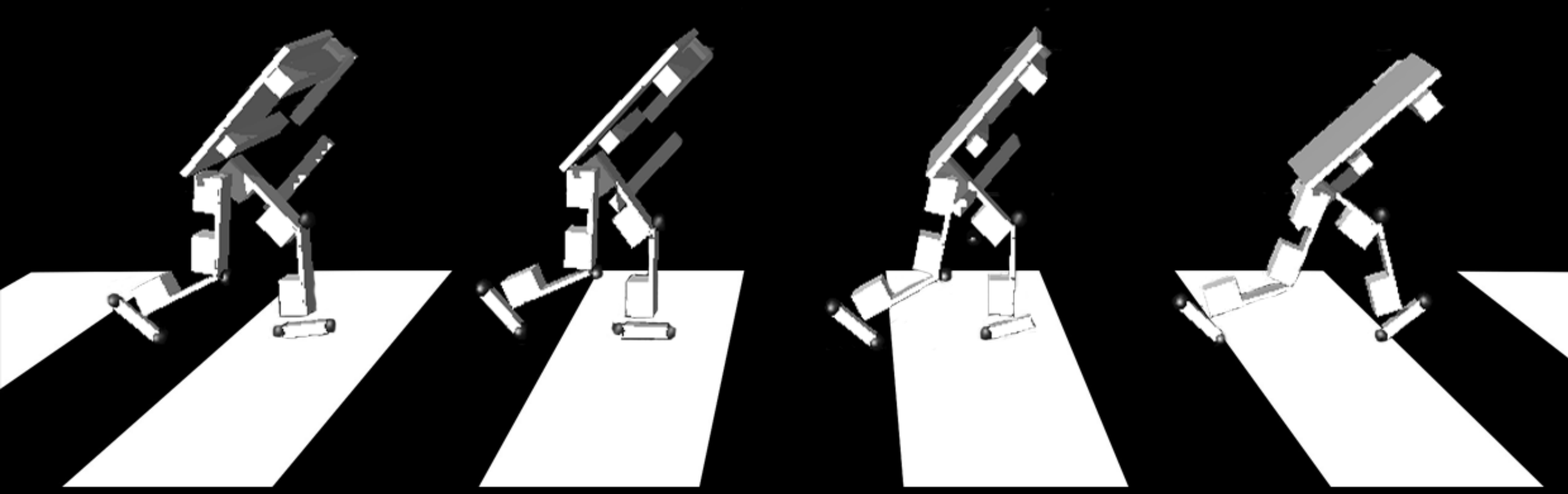
Deep Reinforcement Learning for Bipedal Robots

Divyam Rastogi

Master of Science Thesis



Delft University of Technology

Delft Center for Systems and Control

Deep Reinforcement Learning for Bipedal Robots

Master of Science Thesis

For the degree of Master of Science in Systems and Control at Delft University of Technology

Divyam Rastogi

August 21, 2017

Faculty of Mechanical, Maritime and Materials Engineering (3mE) · Delft University of Technology

Abstract

Reinforcement Learning (RL) is a general purpose framework for designing controllers for non-linear systems. It tries to learn a controller (policy) by trial and error. This makes it highly suitable for systems which are difficult to control using conventional control methodologies, such as walking robots. Traditionally, RL has only been applicable to problems with low dimensional state space, but use of Deep Neural Networks as function approximators with RL have shown impressive results for control of high dimensional systems. This approach is known as Deep Reinforcement Learning (DRL).

A major drawback of DRL algorithms is that they generally require a large number of samples and training time, which becomes a challenge when working with real robots. Therefore, most applications of DRL methods have been limited to simulation platforms. Moreover, due to model uncertainties like friction and model inaccuracies in mass, lengths etc., a policy that is trained on a simulation model might not work directly on a real robot.

The objective of the thesis is to apply a DRL algorithm, the Deep Deterministic Policy Gradient (DDPG), for a 2D bipedal robot. The bipedal robot used for analysis is developed by the Delft BioRobotics Lab for Reinforcement Learning purposes and is known as LEO. The DDPG method is applied on a simulated model of LEO and compared with traditional RL methods like SARSA. To overcome the problem of high sample requirement when learning a policy on the real system, an iterative approach is developed in this thesis which learns a difference model and then learns a new policy with this difference model. The difference model captures the mismatch between the real robot and the simulated model.

The approach is tested for two experimental setups in simulation, an inverted pendulum problem and LEO. The difference model is able to learn a policy which is almost optimal compared to training on a real system from scratch, with only 10 % of the samples required.

Table of Contents

Acknowledgements **ix**

1 Introduction **1**

1-1 Walking Robots . . . 1

1-2 Problem Definition . . . 3

1-3 Research Goal . . . 4

1-4 Approach . . . 4

1-5 Contribution . . . 5

1-6 Thesis Outline . . . 5

2 The Reinforcement Learning Problem **7**

2-1 Introduction . . . 7

2-2 Markov Decision Process . . . 7

2-3 Policy . . . 8

2-4 Return . . . 8

2-5 Value Function . . . 8

2-6 Bellman Equations . . . 9

2-7 Model-free Methods . . . 10

2-7-1 Value Function Based Methods . . . 10

2-7-2 Policy Search Methods . . . 13

2-7-3 Actor Critic Methods . . . 13

2-8 Model-based Methods . . . 14

2-9 Summary . . . 15

3 Deep Neural Networks 17
3-1 Introduction 17
3-2 Building Units 17
3-2-1 Artificial Neuron 17
3-2-2 Activation Functions 18
3-3 Feed Forward Networks 19
3-3-1 Training the network 19
3-3-2 Regularization 19
3-4 Summary 21

4 Deep Reinforcement Learning 23
4-1 Introduction 23
4-2 Deep Q-Learning 23
4-3 Deep Deterministic Policy Gradient 25
4-4 Other Deep RL Algorithms 26
4-4-1 Policy Gradient Methods 26
4-4-2 Guided Policy Search 27
4-5 Comparison of DRL Methods 27

5 DDPG for Ideal Conditions 29
5-1 Introduction 29
5-2 LEO 29
5-3 Learning Parameters for DDPG 31
5-4 Results and Discussion 32
5-5 Summary 34

6 Sample Efficient DDPG for Real World Conditions 35
6-1 Introduction 35
6-2 Notation 35
6-3 Proposed Method 36
6-3-1 Algorithm 38
6-4 Summary 40

7 Sample Efficient DDPG Results 41
7-1 Introduction 41
7-2 Experimental Setups 41
7-2-1 Inverted Pendulum 41
7-2-2 LEO 43
7-3 Learning Parameters 43
7-4 Results and Discussion 45
7-5 Advantages of the approach 52
7-6 Limitations of the approach 53
7-7 Summary 54

8 Conclusion 55

List of Figures

1 Robot LEO . . . 3

2 An Artificial Neuron . . . 18
3 Feed Forward Network . . . 19
4 Comparison of neural networks with and without dropout . . . 20
5 Behaviour of neurons and the associated weights for dropout during training and test runs . . . 21

6 Real LEO and simulated model of LEO . . . 30
7 Cumulative rewards for LEO for different models . . . 33
8 Distance walked by LEO for different models . . . 33

9 Schematic of the proposed method . . . 37
10 Integration of difference model with RL framework . . . 39

11 Inverted Pendulum model . . . 42
12 Training and test errors for training data with and without dropout (LEO). . . . 44
13 Comparison of results for different policies evaluated on the real model of the pendulum . . . 45
14 State trajectories and control input for different policies evaluated on the real model of the pendulum . . . 46
15 State trajectories and control input for successfully learned policies on the ideal and real model of the pendulum . . . 47
16 Comparison of results for different policies evaluated on the real model of LEO . 47
17 Trajectory of centre of mass (COM) of the torso for different policies evaluated on the real model of LEO . . . 48
18 Comparison of state distributions for policies evaluated on the real model and with the difference model for LEO . . . 50
19 Training data for difference model for pendulum and LEO . . . 52

List of Tables

1 Comparison of DRL methods based on sample efficiency 28
2 Comparison between TRPO and DDPG 28

3 Comparison of results for different algorithms 33

4 Pendulum model parameters 42
5 Difference in physical parameters between ideal and real model 43
6 Comparison of performance of different policies evaluated on the real model . . . 48
7 Reduction in error with difference model 51

Acknowledgements

First of all, I would like to start by thanking my supervisors, Jens Kober and Ivan Koryakovskiy, for giving me the opportunity to work on this thesis. This project would not have been possible without their support. They were always ready to provide me with feedback on my work and give me ideas whenever I found myself stuck on something. They were always available to meet me on very short notice if I needed some help.

Next, I would like to thank my friends at TU Delft Milan, Ayush, Salil, Abhimanyu, Sreekar and Siva for their help and assistance during the course of this thesis. I highly appreciate the talks that I had with everyone at different points during the course of the project. Special thanks to Salil for designing the cover of the thesis and also for proof-reading the report and Abhimanyu for coming up with the idea of a Beatles inspired cover page.

Last but not the least, I would like to thank my parents and my sister for all their support and love throughout my life.

Delft, University of Technology
August 21, 2017

Divyam Rastogi

Chapter 1

Introduction

Service robots are likely to take over a certain part of our daily household chores (IFR Statistical Department, 2016). For service robots to integrate in the human environment, they should have robust walking gaits. Humans are highly efficient in the way they walk [1]. Their walking gaits are extremely energy efficient and enable them to walk on any terrain. Despite the many advancements in humanoid robotics in the recent years, walking is a major bottleneck for the robotics industry. A look at the recent DARPA Robotics Challenge, which showcases the state of the art in humanoid robotics, shows how humanoid robots are still incapable of walking robustly even on flat terrains.

So as part of my thesis I would like to tackle the problem of bipedal walking using the technique known as *Reinforcement Learning* (RL). I would like to use Deep Neural Networks (DNN) as function approximators in RL for solving this problem. DNN have shown impressive results for control of high dimensional systems when used with RL [2, 3, 4, 5, 6], the approach known as *Deep Reinforcement Learning* (DRL).

The chapter will give a brief introduction to walking robots and explain the difficulties in designing robust walking controllers. The chapter will further discuss the problems in applying DRL to real systems and the approach that is developed in thesis to tackle this problem.

1-1 Walking Robots

Bipedal walking is one of the most challenging and intriguing controls problem in recent years. It poses a variety of challenges which make it difficult to apply classical control methodologies to it. There are four main reasons which makes bipedal walking control difficult:

- Non-linear dynamics: Bipedal walking is nonlinear and unstable in nature. This makes linear control methodologies unsuitable for such systems.
- Changing dynamics: Each walking cycle consists of two phases: the statically stable *double support phase*, where both feet of the robot touch the ground and the statically

unstable *single support phase*, where only one foot touches the ground. The robot requires control strategies which can help it with the transition between these phases.

- Multivariable system: The interactions and coordination required between different joints is considered a very difficult control problem to solve [7].
- Uncertainty in environment and model: Changing terrains makes it hard to design robust walking controllers. Also, most classical control techniques require an exact model of the system which is difficult to create.

Most traditional approaches to bipedal walking use walking pattern generators based on Zero-Moment Point [8], passive dynamic walkers [9, 10] or use optimization techniques for gait synthesis [11, 12]. The major drawback of these kind of approaches is that it requires an exact model of the robot. An exact model is difficult to design due to model uncertainties like friction, contact discontinuities, noise etc. Also, these approaches work well in domains for which they are defined but do not generalize well to uneven terrains or in the presence of external disturbances [13]. This becomes important since robots are beginning to move out into the cities instead of being limited to factories which are highly controlled environments.

This motivates the choice of using learning control algorithms, like RL, which tries to learn a policy (controller) by trail and error. This allows the robot to interact with its environment and learn how to deal with the uncertainty associated with it in a robust and autonomous manner.

Reinforcement Learning

Reinforcement Learning [14] provides a framework for learning robust control policies without requiring any prior knowledge about the environment or the agent which is trying to accomplish a task. The agent is provided with a reward for every action that it takes, by which it tries to learn a policy which maximizes the expected sum of rewards. Reinforcement Learning is an established technique and has already been successfully applied to learn a variety of robotic tasks [15]. Chapter 2 gives a formal introduction of RL and talks about the various traditional RL algorithms.

LEO

LEO is a 2D bipedal robot developed by the Delft BioRobotics Lab [16]. LEO is specifically designed to learn walking using learning methods - like RL. The hip of LEO is attached to a boom which prevents it from moving sideways, thus restricting the robot to go around in circles. The boom also provides the robot with power. LEO is equipped with 7 actuators (hips, knees, ankles and shoulder). For this thesis, the shoulder is not actuated and kept fixed. Thus, the action space of LEO has dimensionality 6. The robot is shown in Figure 1 and complete model is explained later in Chapter 5.

It has already been shown in literature that it is possible to learn a policy for LEO in the simulated environment but not in the real environment from scratch [17]. The main reason for RL not being able to learn a policy for real LEO is the fact that LEO broke down after every 5 minutes of training, due to falls and application of different torques at high frequency.

This made it difficult to apply RL on it for a longer duration of time to be able to learn a policy. However, Schuitema [17] managed to train a policy on real LEO by reducing the policy learning time through learning via demonstrations. In spite of adding prior knowledge, the method only worked for a reduced state space of LEO with only 3 of the joints actuated through RL while the other 3 were controlled using a pre-programmed controller.

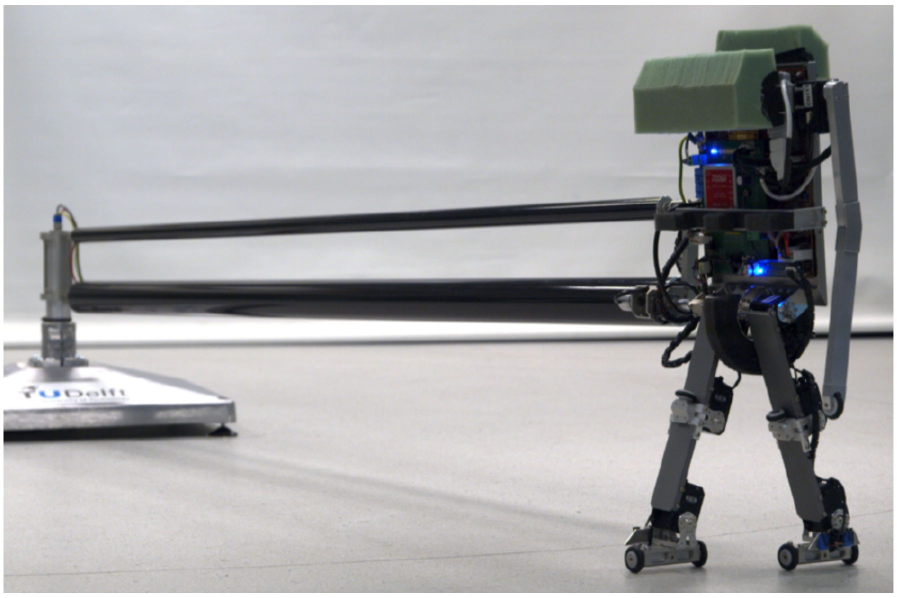


Figure 1: The robot LEO [18]

1-2 Problem Definition

Reinforcement Learning has been successfully applied to bipedal walking problem for only a limited number of cases [19, 20, 21, 22, 23, 24]. Most of these approaches try to incorporate a certain level of prior knowledge into training. They either do it by using a pre-structured policy [20, 21] or by reducing the dimensionality of the controlled system, by decreasing the number of actuated joints [16] or using the symmetry of the robot [24]. Since RL algorithms do not scale well with increasing dimensionality, such measures are introduced to reduce the sample complexity of the algorithm and get traditional RL algorithms to work.

This leads to one of the major drawback of these methods which makes them only applicable to a specific type of robot and, hence, lacking in generality. Thus, it is important to look for a more general class of methods which can be used on a wide variety of bipedal robots. In the light of recent success of DRL methods for control of high dimensional continuous systems, it appears that DRL is a good option to use for control of bipedal robots.

However, most methods of finding an optimal policy using DRL algorithms only exist for simulated environments. The main reason for this is the high sample requirement for these algorithms. In general, DRL algorithms require more than 10^5 samples for their convergence. Those number of samples would be very difficult to obtain on a real system without damaging it. Therefore, the most easy approach to learn a policy for a real system would be to learn a policy on a simulated model and then apply it on the real system. However, an important distinction arises when policies learned on a simulated model are applied to a real system. Due to discrepancies between the simulated model and the real system, policies which perform optimally in simulation generally fail to perform adequately when applied to the real system.

A fair amount of work has been done in this regard, such as improving the simulation to better match the real world. This involves improving the handling of contacts, friction and also improving the physical identification of quantities like masses, lengths and friction coefficients. But this approach has two major disadvantages. Firstly, it requires a fair amount of time to accurately identify these parameters and there is no guarantee that these things would remain the same over the course of a certain time duration. If any of those parameters change, it would require changing the model and tuning the parameters of the control scheme again to figure out the optimal policy. Moreover, if a control scheme is designed which is robust and works for a wide range of model parameters, it might again perform sub-optimally when applied to the real system. Secondly, having a highly complex and descriptive model makes the simulation slow and finding a optimal policy more challenging. Thus, a novel approach needs to be designed which can allow DRL methods to learn optimal policies for real systems, while simultaneously reducing the sample requirement from these systems.

1-3 Research Goal

The goal of the thesis is to investigate the advantages of applying DRL to high dimensional systems and address roadblocks in the application of DRL algorithms to real systems. Therefore, the following research questions are addressed:

- What is the advantage of applying DRL methods to LEO as compared to traditional RL methods like SARSA?
- How to reduce the amount the amount of interaction time required with the real system when learning a policy using DRL methods?

1-4 Approach

The first part of the thesis is focused on implementing a DRL algorithm, the Deep Deterministic Policy Gradient (DDPG), for LEO. The second part of the thesis deals with the challenges of applying DDPG to real systems and how can they be overcome.

One class of RL algorithms which tackle the problem of applying RL algorithms for robotics are known as model-based methods. In model-based RL, training a policy is interleaved with learning a model. The model is learned by generating samples from the real system. The model is then used for generating additional roll-outs which increase the sample efficiency and allow better performance on real systems [14, 25, 26].

This motivates the choice of using a model-based approach with DDPG to improve the performance of a policy on the real system. However, learning a complete dynamic model is not desirable since it requires more data and the quality of the model largely depends on the diversity of samples. To ensure good sample diversity for LEO, it will require the application of varied torques at different configurations which can be dangerous for the robot. Another approach is to learn a locally adaptive dynamic model which can (1) accurately model the behaviour of the system in regions which are most important for the policy, (2) less data intensive to compute and (3) safe for the robot. So, instead of learning a complete dynamic model, a difference model can be learned. This difference model will account for discrepancies between the simulated model and the real system and model the 'simulation gap'. This kind of a approach has been investigated in [27] where a Gaussian process model is used to model the difference. The big drawback of using Gaussian process (GP) models is that they suffer from the curse of dimensionality for large dataset sizes. This will be a problem for LEO. Thus, there is a need to look for a model architecture which can scale well for higher dimensions and large dataset sizes. Recently, forward and inverse dynamic models based on DNN have been successfully learned to model complex systems like a 3D humanoid and 2D walker [28], and a helicopter [29]. The dimensionality of such systems is similar to LEO. This motivates the choice of using a DNN for our approach as well.

Therefore, in this thesis, a DNN model-based approach with DDPG is proposed. A difference model will be learned which can model the 'simulation gap' between the simulated model and the real system. The difference model will be modeled as a DNN. Since the 'simulation gap' only matters in regions of the state space explored by the policy, the difference model will only need to be accurate in those parts of the state space and thus, can be computed efficiently. The proposed approach is generic and can be utilized for a wide variety of robotic tasks.

1-5 Contribution

The main contributions of the thesis are:

- The implementation of a DRL algorithm for the full state space of LEO. This is the first time that an RL algorithm has been shown to learn a policy for a complete state-action space model of LEO.
- The derivation of an iterative approach to tackle the problem of high sample requirement for DRL algorithms. This is achieved through the transfer of policies learned on simulated models optimally to the real system.

1-6 Thesis Outline

The remainder of the thesis is structured as follows:

Chapter 2 provides a formal introduction to RL by explaining the mathematical preliminaries associated with it. The chapter also discusses traditional RL approaches, model-free RL and model-based RL, which forms the basis of most DRL methods.

Chapter 3 introduces DNN and explains its different components. The chapter also discusses a regularization technique which improves the performance of the neural network.

Chapter 4 introduces a few DRL algorithms and provides motivation for choosing a particular algorithm for this thesis.

Chapter 5 presents results for implementing the particular DRL algorithm for LEO and compares its performance with SARSA.

Chapter 6 introduces the proposed approach developed in this thesis to optimally transfer policies learned on simulated models to real systems.

Chapter 7 evaluates the performance of the proposed method on two experimental setups and discusses the results. The chapter also explains the advantages and limitations of this approach.

Chapter 8 presents a summary and conclusion of the work done in the thesis and provides directions for future work.

Chapter 2

The Reinforcement Learning Problem

2-1 Introduction

Reinforcement Learning (RL) is a branch of machine learning which deals with sequential decision making. A RL problem consists of an agent and an environment. The RL agent acts on the environment and gets a scalar reward. The reward is a way of letting the agent know as to how good the action was at that particular instant. The task of the RL agent is to learn an optimal policy - a mapping from states to action, which maximizes the expected sum of rewards. RL is different from supervised learning in the sense that in RL the agent does not know what the right action is at the particular instant and must figure that out based on trial and error. The RL problem is defined in terms of optimal control of a Markov Decision Process (MDP). The other components of an RL problem such as Policy, Value function, and the Bellman Optimality condition will be discussed in this chapter. The chapter will also look into traditional RL approaches, model-free and model-based. The chapter is based on Sutton and Barto [14].

2-2 Markov Decision Process

A Markov Decision Process (MDP) is defined by:

- A set of states, $s \in S$, where S denotes the state space
- A set of actions, $a \in A$, where A denotes the action space
- A scalar reward, r
- Transition probability

$$p(s'|s,a) = Pr(s_{t+1} = s'|s_t = s, a_t = a)$$

- The reward function, $R : S \times S \times A \rightarrow \mathbb{R}$
$$R(s, a, s') = \mathbb{E}(r_t | s_t = s, a_t = a, s_{t+1} = s')$$
- Discount factor, γ

The reward function and the transition probabilities define the most important aspects of a MDP and are usually considered unknown in a general reinforcement learning setting.

2-3 Policy

The policy is defined as a mapping from states to actions. The policy could be deterministic, and depend only on the state, $\pi(s)$, or stochastic, $\pi(a|s)$, such that it defines a probability distribution over the actions, given a state.

2-4 Return

The objective of a RL agent is to choose a policy which maximizes the expected sum of rewards. The sum of rewards is called as the *return*, G_t, which is given as,

$$G_t = r_t + r_{t+1} + r_{t+2} + \ldots + r_{N-1}$$

where, N denotes the end of an episode, t is the time index.

For continuing tasks, the discounted return is defined, which is given by

$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$$

where $\gamma \in [0, 1)$ is the discount factor.

2-5 Value Function

In order to be able to decide what action to take at a particular instant, it is important for the agent to know how good it is for the agent to be in a particular state. A way of measuring the "goodness" of a state is the *value function.* It is defined as the expected sum of rewards that the agent will receive while following a particular policy π from a particular state s. The *value function*, $V_\pi(s)$ for policy π is given by

$$V_\pi(s) = \mathbb{E}_\pi(G_t | s_t = s) = \mathbb{E}_\pi\left(\sum_{k=0}^{\infty} \gamma^k r_{t+k} | s_t = s\right)$$

Similarly, an *action value function*, also known as the *Q-function*, can be defined which is the expected sum of rewards while taking action a in state s and, thereafter, following policy π. The *action value function*, $Q_\pi(s, a)$ is given by

$$Q_\pi(s, a) = \mathbb{E}_\pi(G_t | s_t = s, a_t = a) = \mathbb{E}_\pi\left(\sum_{k=0}^{\infty} \gamma^k r_{t+k} | s_t = s, a_t = a\right) \tag{2-1}$$

2-6 Bellman Equations

The Bellman equations formulate the problem of maximizing the expected sum of rewards in terms of a recursive relationship of a value function. A policy π is considered better than another policy π' if the expected return of that policy is greater than π' for all $s \in S$, which implies, $V^{\pi}(s) \geq V^{\pi'}(s)$ for all $s \in S$. Thus, the optimal value function, $V_*(s)$ can be defined as,

$$V_*(s) = \max_{\pi} V_\pi(s), \;\forall\; s \in S.$$

Similarly, the optimal *action value function*, $Q_*(s,a)$ can be defined as,

$$Q_*(s,a) = \max_{\pi} Q_\pi(s,a), \;\forall\; s \in S, \; a \in A.$$

Also, for an optimal policy, the following equation can be written,

$$V_*(s) = \max_{a \in A(s)} Q_{\pi^*}(s,a) \tag{2-2}$$

Expanding Eqn. (2-2) with (2-1),

$$\begin{aligned} V_*(s) &= \max_a \mathbb{E}_{\pi^*}(G_t | s_t = s, a_t = a) \\ &= \max_a \mathbb{E}_{\pi^*}\left(\sum_{k=0}^{\infty} \gamma^k r_{t+k} | s_t = s, a_t = a\right) \\ &= \max_a \sum_{s'} p(s'|s,a)[R(s,a,s') + \gamma V_*(s')] \end{aligned} \tag{2-3}$$

Equation (2-3) is known as the Bellman optimality equation for $V_*(s)$. The Bellman optimality equation for Q_* is

$$\begin{aligned} Q_*(s,a) &= \mathbb{E}(r_t + \gamma \max_{a'} Q_*(s_{t+1}, a') | s_t = s, a_t = a) \\ &= \sum_{s'} p(s'|s,a)[R(s,a,s') + \gamma \max_{a'} Q_*(s', a')] \end{aligned}$$

If the transition probabilities and the reward functions are known, the Bellman optimality equations can be solved in an iterative fashion. This approach is known as *Dynamic programming.* The algorithms which assume these probabilities to be known or estimate them online are collectively known as *model-based* (Section 2-8) algorithms.

But for most other algorithms, they assume the probabilities are not known and they estimate the policy and value function by performing rollouts on the system. These methods are known as *model-free* algorithms (Section 2-7). Monte Carlo, temporal difference and policy search methods are the most common model-free algorithms used.

2-7 Model-free Methods

Model-free methods can be applied to any reinforcement learning problem since they do not require any model of the environment. Most model-free approaches either try to learn a value function and infer an optimal policy from it (Value function based methods) or directly search in the space of the policy parameters to find an optimal policy (Policy search methods).

Model-free approaches can also be classified as being either *on-policy* or *off-policy*. On-policy methods use the current policy to generate actions and use it to update the current policy while off-policy methods use a different exploratory policy to generate actions as compared to the policy which is being updated. The following subsections look at various model-free algorithms used, both value function based and policy search methods.

2-7-1 Value Function Based Methods

Monte Carlo Methods

Monte Carlo methods work on the idea of generalized policy iteration (GPI). The GPI is an iterative scheme and is composed of two steps. The first step tries to build an approximation of the value function based on the current policy, known as the *policy evaluation step*. In the second step, the policy is improved with respect to the current value function, known as the *policy improvement step*. In Monte Carlo methods, to estimate the value function, rollouts are performed by executing the current policy on the system. The accumulated reward over the entire episode and the distribution of states encountered is used to form an estimate of the value function. The current policy is then estimated by making it directly greedy with respect to the current value function. Using these two steps iteratively, it can be shown that the algorithm converges to the optimal value function and policy. Though Monte Carlo methods are straightforward in their implementation, they require a large number of iterations for their convergence and suffer from a large variance in their value function estimation.

Temporal Difference Methods

Temporal difference (TD) builds on the idea on the GPI but differs from the Monte Carlo methods in the policy evaluation step. Instead of using the total accumulated reward, the methods calculate a temporal error, which is the difference of the new estimate of the value function and the old estimate of the value function, by considering the reward received at the current time step and use it to update the value function. This kind of an update reduces the variance but increases the bias in the estimate of the value function. The update equation for the value function is given by:

$$V(s) \leftarrow V(s) + \alpha[r + \gamma V(s') - V(s)]$$

where, α is the learning rate, r is the reward received at the current time instant, s' is the new state and s is the old state.
Thus, temporal difference methods update the value function at each time step, unlike the Monte Carlo methods which wait till the episode has ended to update the value function.

Two TD algorithms which have been widely used to solve RL problems are *SARSA (State-Action-Reward-State-Action)* and *Q-Learning.*

SARSA is an on-policy temporal difference algorithm which tries to learn an action value function instead of a value function. The policy evaluation step uses the temporal error for the action value function, which is similar to the value function. The algorithm is summarized below:

Algorithm 1 SARSA

```
Initialize $Q(s, a)$ randomly
repeat
    Observe initial state $s_1$
    Select an action $a_1$ using policy derived from $Q$ (e.g. $\epsilon$-greedy)
    for t=1:T do
        Carry out action $a_t$
        Observe reward $r_t$ and new state $s_{t+1}$
        Choose next action $a_{t+1}$ using policy derived from $Q$ (e.g. $\epsilon$-greedy)
        Update $Q$ using
            $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$
    end for
until terminated
```

Watkins [30] introduced an off-policy temporal difference algorithm known as Q-learning. Q-learning is off-policy since the action is chosen with respect to an ϵ-greedy policy while the Q-function is updated by using a policy which is directly greedy with respect to the current Q-function. Q-learning differs from SARSA in the way the Q-function is updated. The update rule is given by

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \max_a \gamma Q(s', a) - Q(s, a)] \tag{2-4}$$

The algorithm is summarized below:

Algorithm 2 Q-learning

```
Initialize $Q(s, a)$ randomly
repeat
    Observe initial state $s_1$
    for t=1:T do
        Select an action $a_t$ using policy derived from $Q$ (e.g. $\epsilon$-greedy)
        Carry out action $a_t$
        Observe reward $r_t$ and new state $s_{t+1}$
        Update $Q$ using (2-4)
    end for
until terminated
```

Monte Carlo methods use the total reward for value function update whereas TD methods use the reward at the current step. A way to combine both approaches would be to use the returns obtained after every certain number of steps and take a weighted average of those returns. This gives the TD(λ) algorithm.

Each state is associated with an additional variable known as the *eligibility trace*, $e_t(s) \in \mathbb{R}^+$. The eligibility trace indicates how much learning should be imparted to each state at every time step and it depends on how much the state has been visited in the current past. The eligibility trace for each state is updated at each time step by the following rule:

$$e_t(s) = \begin{cases} \gamma\lambda e_{t-1}(s), & \text{if } s \neq s_t \\ \gamma\lambda e_{t-1}(s) + 1, & \text{if } s = s_t \end{cases}$$

where λ is known as the *trace-decay parameter*.
This is referred to as the *accumulating* form of the eligibility trace. The value function is updated by either a SARSA or Q-learning update, weighted by the eligibility traces.

If the value functions are stored in a tabular form, then these methods are only applicable to low dimensional tasks. For most useful applications, function approximators are used for storing value functions which are less memory extensive and can generalise better to high dimensional tasks.

Function Approximators

The easiest way to save the values of a value function for different states is in a tabular form. However, if the state space of the problem is large, it becomes impossible to store all the values in a tabular format. The reason is the amount of memory that is required to store all the data. Additionally, when trying to look up some value for a particular state, it will require an entire sweep of the table which is extremely computationally expensive. Also, if the state space is continuous, then a tabular format is impossible.

To overcome this problem, function approximators are used to store a value function. The value function is parameterized by a vector $\theta = (\theta_1, \theta_2, \ldots, \theta_n)^T$ and is denoted as $V(s; \theta^V)$. The function approximator can be thought of as a mapping from a vector θ in $\mathbb{R}^n$ to the space of the value function. Since the number of parameters of the functional approximator is less than the number of state values, changing a value of a certain parameter results in changes of the value function in many regions of the state space. This helps function approximators to generalize better in lesser number of training samples.

A wide variety of function approximators have been used in literature, like tile coding, radial basis functions or neural networks. Tile coding and radial basis functions were a popular form of function approximators before the popularity of neural networks. Nowadays, neural networks are widely used for this purpose. The advantage of using neural networks over tile coding or radial basis functions is that neural networks are capable of more complex representations of value function with lesser number of parameters as compared to other methods. This reduces training time for RL algorithms for high dimensional systems and is less memory extensive. Despite the advantages of neural networks, using neural networks as function approximators require certain additional 'tricks' to enable them to be effectively applied with RL (Section 4-2). They also do not provide any theoretical guarantees on convergence.

2-7-2 Policy Search Methods

Policy search methods are another class of RL algorithms which use parameterized policies, π_{θ^π}, with θ^π being the parameter vector. These methods search in the parameter space, $\Theta, \theta^\pi \in \Theta$ for an optimal policy. The policy is evaluated by performing rollouts from the current policy and calculating the reward. The parameters of the policy are then updated in the direction of increasing expected return using gradient descent. The update rule for the parameters of the policy can be written in terms of the expected return, J, as

$$\theta^\pi_{t+1} = \theta^\pi_t + \alpha \nabla_{\theta^\pi} J \quad , \quad J = \mathbb{E}_\pi\left(\sum_{k=0}^{\infty} \gamma^k r_k\right)$$

The way the gradient is defined and evaluated gives rise to a wide variety of policy search approaches [31].

Policy search has better convergence properties and can learn stochastic policies which are not possible with value based approaches [32]. The major drawback of policy search algorithms is their policy evaluation step, which suffers from a large variance and, thus, can be slow in learning good policies. This results in a large number of interactions with the environment which makes it undesirable for tasks which involve actual robots. To overcome these challenges, policy search methods sometimes use pre-structured policies, such as motor primitives [33], which are better suited for certain tasks. Additional information such as human demonstration could also be used to initialize the parameters of the policy, which leads to faster convergence.

2-7-3 Actor Critic Methods

Actor critic methods are TD methods which store the policy explicitly. The policy is known as the *actor*, as it provides the action in a given state. The value function acts as the *critic* since it evaluates the policy based on the temporal difference error. The policy is updated based on this critic. The actor critic method is mostly on-policy, but off-policy actor critic have been introduced in the literature [34].

Actor critic methods provide better convergence properties than the previous TD methods [35]. It also makes the computation of the action easier, especially for continuous tasks. Actor critic methods can also learn stochastic policies.

Actor critic algorithms can either store the actor and critic in a tabular form or can use function approximators, which is the case with most robotic applications. When using function approximators, the policy is updated similar to policy search methods, except that the critic decides the direction of gradient ascent instead of the expected return. This helps to reduce the variance of these types of algorithms. The stochastic policy gradient theorem [36] defines the gradient of the expected return, J, as

$$\begin{aligned}\nabla_{\theta^\pi} J &= \int_{\mathbf{S}} \rho_\pi(s) \int_{\mathbf{A}} \nabla_{\theta^\pi} \pi(a|s;\theta^\pi) Q(s,a;\theta^Q) da ds \\ &= E_{s\sim\rho_\pi, a\sim\pi(\cdot;\theta^\pi)}[\nabla_{\theta^\pi}\pi(a|s;\theta^\pi) Q(s,a;\theta^Q)]\end{aligned}$$

where $\rho_\pi(s)$ is the state distribution under policy π, θ^π is the parameter vector for the policy π and θ^Q is the parameter vector for Q-function.

The parameters of the action value function, $Q(s, a; \theta^Q)$, are updated using temporal difference learning. These types of algorithms are known as *stochastic actor critic algorithms*.

Silver et al. [37] introduced the deterministic policy gradient (DPG) theorem which defines the gradient of the expected return, J, subject to a deterministic policy. For the deterministic case, the gradient of the expected return, J, can be written as

$$\begin{aligned}\nabla_{\theta^\pi} J &= \int_{\mathbf{S}} \rho_\pi(s) \nabla_{\theta^\pi} \pi(s; \theta^\pi) \nabla_a Q(s, a; \theta^Q)|_{a=\pi(s;\theta^\pi)} ds \\ &= E_{s\sim\rho_\pi}[\nabla_{\theta^\pi} \pi(s; \theta^\pi) \nabla_a Q(s, a; \theta^Q)|_{a=\pi(s;\theta^\pi)}]\end{aligned}$$

Thus, the *off-policy deterministic actor critic* (OPDAC) algorithm was developed, which updates the actor and critic as

$$\begin{aligned}\delta_t &= r_t + \gamma Q(s_{t+1}, \pi(s_{t+1}; \theta^\pi); \theta^Q) - Q(s_t, a_t; \theta^Q) \\ \theta^Q_{t+1} &= \theta^Q_t + \alpha \delta_t \nabla_{\theta^Q} Q(s_t, a_t; \theta^Q) \\ \theta^\pi_{t+1} &= \theta^\pi_t + \alpha_\theta \nabla_{\theta^\pi} \pi(s_t; \theta^\pi) \nabla_a Q(s_t, a_t; \theta^Q)|_{a=\pi(s;\theta^\pi)}\end{aligned}$$

where δ_t is the *temporal difference error* and α, α_θ are the learning rates.

The DPG gradients can be computed more efficiently than the stochastic case and these algorithms show significantly better performance than their stochastic counterpart. A version of the OPDAC algorithm which uses neural networks as function approximators for the actor and critic, known as Deep Deterministic Policy Gradient (DDPG) has also been implemented (Section 4-3).

2-8 Model-based Methods

All the methods that have been discussed so far do not possess any knowledge of the environment. Model-based methods try to utilize some information of the environment for planning. Planning with a model can substantially improve the sample efficiency of the algorithm which makes it popular in robotic applications. There are two approaches to model-based learning: one is to build a model of the environment from first principles and use it for learning the policy or value function. The other method is to learn a model of the environment by performing experiments on the system.

The problem with starting with a first principles model is that if the model is not accurate, the policy learned is likely to be sub optimal when applied on the actual setup. It might also happen that the policy just out right fails to do anything. Thus, most successful model-based approaches try learning a model and then using the model to accelerate the learning.

A variety of forward models have been proposed in literature for planning: locally linear models [38, 39], Gaussian Process models [26, 27], Gaussian mixture models [4], DNN [29, 28] and so on. The efficiency of the models depends on how much memory it requires and how does it scale with increasing dimensionality. Thus, the choice of the model is largely dependent on the problem. Apart from forward models, a few architectures which combines planning and direct RL have also been proposed.

Dyna-Q

Dyna-Q [14] is one such architecture which tries to integrate model learning, planning and direct RL. The algorithm learns a model by seeing which states are visited and what reward it gets in a particular state and uses the information to update the transition probabilities and the reward function. The model is then used to perform imaginative rollouts starting from states which have already been visited. The rollouts are then used to update the Q-function, similar to Q-learning. The model learned is again stored in a tabular form and thus this type of an architecture is only possible for low dimensional tasks.

Real Time Model Based Architecture

Real time model based architecture (RTMBA) [25] tries to parallelize the processes of model learning, planning and direct RL. The algorithm tries to improve on DYNA-Q by allowing it to take actions continually in real time, without waiting for planning or learning processes to complete. The algorithm works by having three threads: model learning, planning and action thread. The model learning waits for the action thread to do an action and then updates the model. The planning thread takes the most recent model from the model learning thread and the most recent state from the action thread and plans rollouts from that state. The action thread uses the Q-function to calculate actions. RTMBA is able to perform better than Dyna-Q in terms of sample efficiency and learning time.

2-9 Summary

This chapter introduced the basics of Reinforcement Learning by explaining many important concepts associated with it. Various popular approaches within traditional RL were also discussed. In general, model-free methods are preferred for simulated environments since they do not need any information about the environment. But, for most real systems, where sample efficiency becomes a priority, model-based methods are preferred. As is it more beneficial to learn a model instead of just deriving a model from first principles, the kind of model to be learned becomes critical. Linear models are easy to compute and update, but they are unable to capture fast and contact-rich dynamics, which is essential for the case of LEO. Gaussian models provide the benefit of being stochastic, but they suffer from the curse of dimensionality. In case of LEO, this is again a problem. DNN, on the other hand, have been shown to learn complex models for high dimensional systems and can handle contact-rich dynamics. They also scale well with increasing data size. Thus, DNN is a promising model architecture to learn for LEO. The next chapter will introduce the basics of DNN and introduce a regularization technique which improves the prediction accuracy of DNNs.

Chapter 3

Deep Neural Networks

3-1 Introduction

Deep Neural Networks are the driving force behind some of the major developments in artificial intelligence and machine learning in the recent years. They are at the heart of most interesting products like self driving cars [40], image recognition systems [41], speech recognition systems [42] and autonomous robots [2, 6, 3, 5]. They have achieved state-of-the-art performances in these tasks and are becoming ubiquitous for these purposes.

DNN provides a general framework for approximating non-linear functions from training examples. These functions can be real valued, discrete valued or vector valued. DNN are so successful due to their ability to learn from data and extract critical features without having to handcraft them.

This chapter will give a brief introduction about the neural network architecture, the algorithms used in training the neural network and will introduce a regularization technique, *dropout*, which improves the prediction accuracy of the neural networks.

3-2 Building Units

Artificial neural networks are loosely modeled on how the human brain works. The human brain is a complex web of interconnected neurons which have the ability to extract critical information from the inputs provided and produce a corresponding output (signal). Similarly, an artificial neural network consists of a set of interconnected neurons, arranged in layers, where each neuron takes some real valued inputs and gives out a real valued output.

3-2-1 Artificial Neuron

Figure 2 shows an artificial neuron. The neuron takes a vector of inputs x and calculates the weighted sum of the inputs, with the weights denoted by w. The weighted sum is added to a *bias* term, b, and is passed through an activation function, f, to produce the output of the neuron. The complete equation for a neuron can be written as

$$y_i = f(\sum_j x_j w_{i,j} + b_i) \tag{3-1}$$

where j is the number of inputs to the neuron.

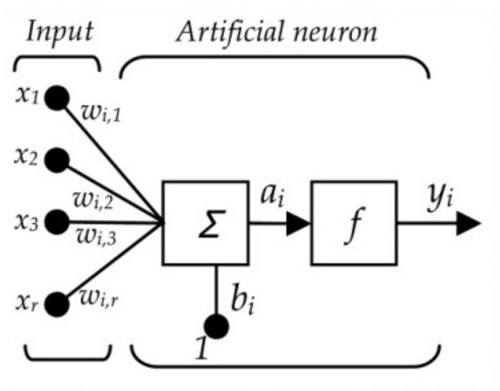


Figure 2: An Artificial Neuron [43]

3-2-2 Activation Functions

The activation function introduces non-linearity into the output of a neuron. This helps the network to learn non-linear representations from the training data. A number of activation functions have been introduced in literature, like *sigmoid*, *hyperbolic tangent* and a *rectified linear unit*. The commonly used activation functions are explained below:

- Sigmoid:

$$f(x) = \frac{1}{1 + e^{-x}}$$

- Hyperbolic Tangent:

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

- Rectified Linear Unit (ReLu):

$$f(x) = \max(0, x)$$

From the activation functions discussed above, *ReLu* provides the best performance over a wide range of tasks, as it does not suffer from saturation as compared to the other two functions.

3-3 Feed Forward Networks

The most common architecture used in DNN is the *Feed-Forward* neural network. Figure 3 shows an example of a feed-forward neural network. In this framework, the neurons are arranged in layers. This architecture usually has three kinds of layers: an *input layer*, a few *hidden layers* and an *output layer.* The flow of information takes place from the input layer to the hidden layers and finally to the output layer which computes the final output. Each neuron in the different layers performs the same computation as shown in Eqn. 3-1.

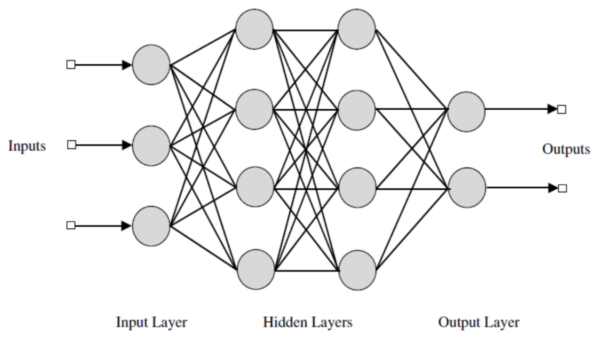


Figure 3: Feed Forward Network [44]

3-3-1 Training the network

The weights of the neural network are trained using the technique known as *backpropagation.* The idea in this approach is to start with a random initialization of the weights and calculate the output for a given input. The error between the generated output and the actual output is used to update the weights of the network using a gradient descent algorithm. Since the weights are first updated of the output layer and then backpropagated into the network, the technique is known as *backpropagation.*

In the recent years, *stochastic gradient descent* has been a popular choice for training the weights of a neural network. A few improved variants of stochastic gradient descent has also been proposed, like *ADAM* [45] and *AdaGrad* [46]. The advantage of using these methods over vanilla gradient descent is that these methods can vary the learning rates based on the distribution of the training data. This reduces the need for careful choice of a learning rate which in turn allows the algorithm to converge faster.

3-3-2 Regularization

Regularization is a technique used in machine learning to prevent the problem of over-fitting in statistical models. The need to introduce regularization is to improve the generalizability of the network to unseen data. The classical approach to solving the problem of over-fitting is to use L1 or L2 regularization. The main idea in this method is to introduce a certain penalty term for the weights, w, in the cost function. Another technique which helps to avoid the problem of over-fitting and performs better than normal regularization is *Dropout* [47].

Dropout

Dropout is a technique used to prevent the problem of over-fitting in large neural networks. The main idea in dropout is to remove certain neurons and their connections in the neural network during training. Thus, at the time of training, dropout samples from a certain number of 'thinned' networks. This prevents different parts of the neural network to specifically adapt to predict certain features of the training data. At test time, however, all of the neurons, with smaller weights, are used to predict the output, which tends to average the effect of dropping neurons in the training phase.

An analogy which is provided for why dropout works is given in the following way: Suppose, a group of 10 people is tasked to carry out work on a building. In general, each one of those people could get accustomed to doing a certain part of the work and get "very good" at it. If a scenario comes where that person is not present to do that work anymore, none of the other persons would be able to carry out his part of the work. Contrary to it, if only a random number of people are allowed to carry out work on the building on a given day, each person could get equally good at performing all the different tasks since no person is guaranteed to arrive on a given day. Mathematically, this is equivalent to saying that the network can generalize to scenarios which it has not encountered before and can result in better performance to unseen data.

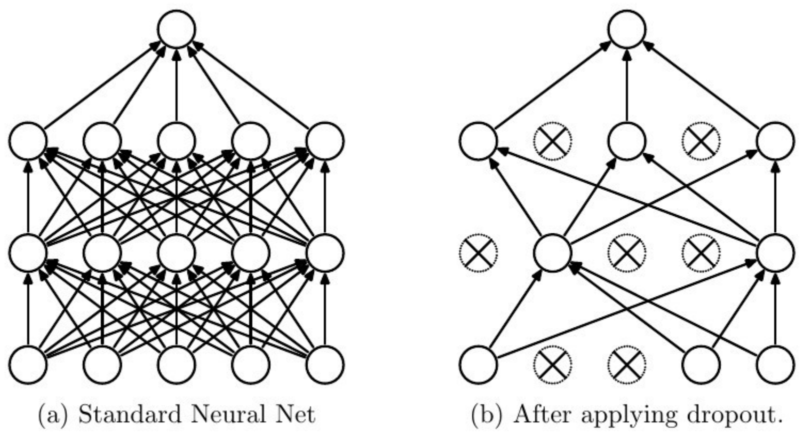


Figure 4: Left: Standard neural network without dropout. Right: Neural network after applying dropout [47].

Figure 4 shows a DNN with two hidden layers. The left neural network is a standard feed-forward neural network. The right neural network is the same network with dropout. As seen from the right network, certain neurons and their connections in the training phase are removed. The neurons to be removed are chosen randomly with a certain probability, p. So, in the training phase, each neuron is present in the network with a probability, $1 - p$, where p can be constant or same for different layers. In general, the probability p is taken to be close to 0.5 for the hidden layers and close to 1 for the input layer. At test time, however, none of the neurons are dropped. To ensure that the complete network approximately predicts the same output as one of the 'thinned' networks, the connection weights of the neuron during test time are scaled by p as shown in Figure 5.

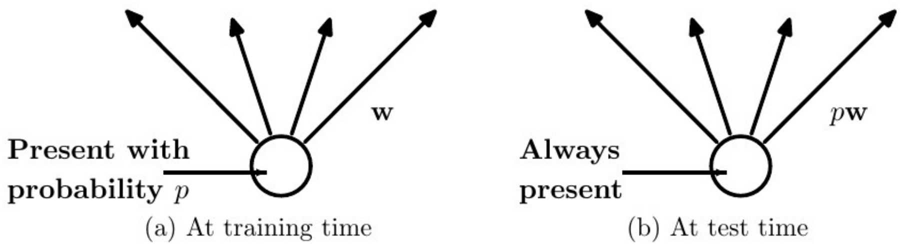


Figure 5: Left: At training time, every neuron is present with probability p. Right: At test time, every neuron is present with probability 1 and the corresponding weights are scaled by p [47].

Dropout provides major improvements over other regularization techniques and has been proven to achieve state-of-the-art performance in a variety of large datasets. In the model learning part of the thesis, dropout is used to improve the generalizability of our network since our training data would mostly be clustered in certain regions of the state space and dropout could help in improving the performance of the model in regions not properly represented in the training data.

3-4 Summary

DNNs are general purpose function approximators which can learn from examples without the need to provide any hand crafted features. They also work well for real time systems since their training algorithm can be massively parallelized by using Graphical Processing Units (GPU). Due to their ability to scale to high dimensions and fast training times, neural networks will be used with RL and also in the model learning part of the thesis (Chapter 6).

Chapter 4

Deep Reinforcement Learning

4-1 Introduction

Deep Reinforcement Learning (DRL) has received a lot of interest among the AI community in the last couple of years. Deep Reinforcement Learning refers to the use of Deep Neural Networks as function approximators for value functions or policy in a RL framework. DRL has achieved human level performance for playing Atari games [2] and for controlling 3D locomotion tasks with high dimensional state space [6, 3]. The fact that DRL can handle high dimensional state and action space makes it extremely suitable for the purpose of controlling the walking motion of a biped robot.

DRL has been successfully applied to policy gradient methods [5, 6], Q-learning [2, 48] and actor critic methods [3, 49]. This chapter will give a detailed explanation of Q-learning in the DRL framework and introduce a deep actor critic algorithm, Deep Deterministic Policy Gradient (DDPG). The chapter will also give an overview of other state-of-the-art DRL algorithms and provide a motivation for using the DDPG for our task by comparing the results of these algorithms on the standard MuJoCo environment [50] and the Atari 2600 games on the Arcade Learning Environment [51].

4-2 Deep Q-Learning

Q-Learning (Section 2-7-1) has been a widely used algorithm for model-free reinforcement learning. It was initially considered that Q-learning was unstable when using with neural networks and thus, most applications of Q-learning were limited to tasks with small state spaces. In [2], it was shown that Q-learning could be used with DNN and the algorithm showed human level performance on seven Atari 2600 games using only raw image pixels as the input.

In the same paper, the Q-function is parameterized with a neural network with weights θ^Q and the algorithm is referred to as *Deep Q-Learning* (DQN). The network is trained by minimizing a loss function at every iteration i, given by

$$L_i(\theta_i^Q) = \mathbb{E}_{s\sim\rho_{\pi(\cdot)},a\sim\pi(\cdot)}\Big(y_i - Q(s,a;\theta_i^Q)\Big)^2$$

where $y_i = \mathbb{E}_{s'\sim\varepsilon}[r + \gamma \max_{a'} Q(s',a';\theta_{i-1}^Q)|s,a]$ is the target at iteration i, $\pi(s|a)$ is the behaviour policy, $\rho_{\pi(\cdot)}$ is the distribution of states under policy $\pi(s|a)$ and ϵ refers to the environment.
To minimize the loss function, the gradient of the loss function is computed with respect to the weights and is given by

$$\nabla_{\theta_i^Q} L(\theta_i^Q) = \mathbb{E}_{s\sim\rho_{\pi(\cdot)},a\sim\pi(\cdot),s'\sim\varepsilon}\Big[\Big(r + \max_{a'} Q(s',a';\theta_{i-1}^Q) - Q(s,a;\theta_i^Q)\Big)\nabla_{\theta_i^Q} Q(s,a;\theta_i^Q)\Big] \tag{4-1}$$

The loss function is minimized using *stochastic gradient descent.* The behaviour policy is an ϵ-greedy policy to ensure sufficient exploration.

The key aspect which makes DQN work is the use of *experience replay.* In general, the sampled trajectories from the environment are temporally correlated and if these trajectories are used to train the network it would lead to over fitting in the network and the network would not be able to learn effectively. In order to break the temporal correlation of data points while training, the agent's experience at each time step, (s_t, a_t, r_t, s_{t+1}), is saved in a replay buffer. At every instant, a fixed number of samples are extracted from the replay buffer at random and used to train the network. This makes the network to learn better. The complete algorithm is given below.

Algorithm 3 Deep Q learning with Experience Replay

```
Initialize replay memory D
Initialize action value function Q with random weights
repeat
    Observe initial state $s_1$
    for t=1:T do
        Select an action $a_t$ using $Q$ (e.g. $\epsilon$-greedy)
        Carry out action $a_t$
        Observe reward $r_t$ and new state $s_{t+1}$
        Store transition $(s_t, a_t, r_t, s_{t+1})$ in replay buffer D
        Sample random transitions $(s_j, a_j, r_j, s_{j+1})$ from D
        Calculate target for each transition
        if $s_{j+1}$ is terminal then
            $y_j = r_j$
        else
            $y_j = r_j + \gamma \max_{a'} Q(s_{j+1}, a'; \theta)$
        end if
        Train the Q network on $(y_j - Q(s_j, a_j; \theta))^2$ using (4-1)
    end for
until terminated
```

The one major drawback of the above algorithm is the need to calculate the maximum over actions and this prohibits the use of of the above algorithm for tasks with continuous actions spaces. To deal with continuous action spaces, an actor critic algorithm was developed which uses the Q-function as the critic and updates the policy using the Deterministic Policy Gradient introduced in Section 2-7-3.

4-3 Deep Deterministic Policy Gradient

The actor critic framework, Deep Deterministic Policy Gradient [3], provides an improvement to DQN discussed above by making it tractable for continuous actions spaces. Both the actor and the policy are described by two separate neural networks. As the name suggests, the algorithm uses the deterministic policy gradient theorem from [37] for policy update. This is also an off-policy algorithm.

It was observed that directly updating the actor and critic parameters by taking the temporal difference as in Eqn. (4-1) resulted in the learning to diverge sometimes. The DDPG algorithm introduces a couple of improvements to the traditional actor critic to make it feasible with neural networks. The first one was already used in DQN, the *experience replay*, and the other is the use of *target networks*. Target networks are two separate networks which are copies of the actor and critic networks. They are used to calculate the target y in the temporal difference error (Eqn. 4-2). The target networks are updated very slowly to track the learned networks. This converts the problem of learning the optimal Q-function into a supervised learning problem which improves the stability of the algorithm immensely.

The update for the critic is given by the standard Q-learning update of (4-1) by taking targets y to be

$$y = \mathbb{E}_{s' \sim \varepsilon}[r + \gamma Q'(s', \mu'(s'; \theta^{\mu'}); \theta^{Q'})] \tag{4-2}$$

where $Q'(s, a; \theta^{Q'})$ and $\mu'(s; \theta^{\mu'})$ refer to the target networks for the critic and the actor with weights $\theta^{Q'}$ and $\theta^{\mu'}$.

The loss function is defined as

$$L = \mathbb{E}_{s \sim \rho_\beta, a \sim \beta}\Big(y - Q(s, a; \theta^Q)\Big)^2 \tag{4-3}$$

where $Q(s, a; \theta^Q)$ refer to the critic network with weights θ^Q and β is the behaviour policy.

The policy is updated by using the deterministic policy gradient theorem from [37] given by

$$\nabla_{\theta^\mu} J = \mathbb{E}_{s \sim \rho_\beta}[\nabla_a Q(s, a; \theta^Q)|_{s, a=\mu(s;\theta^\mu)} \nabla_{\theta^\mu} \mu(s; \theta^\mu)|s] \tag{4-4}$$

where $\mu(s; \theta^\mu)$ refers to the the actor with weights θ^μ.

The complete DDPG algorithm is given below:

Algorithm 4 Deep Deterministic Policy Gradient

```
Initialize replay memory $D$
Initialize critic network $Q$ and actor network $\mu$ with random weights $\theta^Q$ and $\theta^\mu$ respectively
Initialize target networks $Q'$ and $\mu'$ with weights $\theta^{Q'} \leftarrow \theta^Q, \theta^{\mu'} \leftarrow \theta^\mu$
repeat
    Observe initial state $s_1$
    Initialize random process $\mathcal{N}$
    for t=1:T do
        Select an action $a_t = \mu(s_t; \theta^\mu) + \mathcal{N}$
        Carry out action $a_t$
        Observe reward $r_t$ and new state $s_{t+1}$
        Store transition $(s_t, a_t, r_t, s_{t+1})$ in replay buffer $D$
        Sample random transitions from $D$
        Calculate target for each transition using (4-2)
        Train the critic network using stochastic gradient descent by minimizing (4-3)
        Update actor network using (4-4)
        Update the target networks using
            $\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$
            $\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$
    end for
until terminated
```

4-4 Other Deep RL Algorithms

4-4-1 Policy Gradient Methods

Policy Gradient Methods are an effective form of policy optimization for tasks with continuous state space and action spaces. The policy gradient methods are on-policy methods which execute a certain number of trajectories $\{\tau_i\}_{i=1}^N$ using the current policy and then use the transition data $(s_t^i, a_t^i, s_{t+1}^i)$ for each trajectory $\{\tau_i\}$ to update the policy.

Trust Region Policy Optimization

Trust Region Policy Optimization (TRPO) works by minimizing a certain objective function which in turns ensures monotonic improvements to the policy. At each iteration, the following constrained optimization problem is solved,

$$\max_{\theta^\pi} \mathbb{E}_{s\sim\rho_{\pi(\cdot)}, a\sim\pi(\cdot;\theta_i^\pi)} \left(\frac{\pi(a|s;\theta^\pi)}{\pi(a|s;\theta_i^\pi)} Q_\pi(s,a) \right) \tag{4-5}$$
$$\text{subject to } \mathbb{E}_{s\sim\rho_{\pi(\cdot)}} \Big(D_{KL}(\pi(\cdot\,|s;\theta_i^\pi)||(\pi(\cdot\,|s;\theta^\pi) \Big) \leq \delta$$

where $\pi(a|s;\theta_i^\pi)$ denotes the policy at iteration i, $\rho_{\pi(\cdot)}$ denotes the state visitation frequency under policy $\pi_{\theta_i^\pi}$, and δ is the step size which decides how much the new policy is allowed to deviate as compared to the old policy.

The algorithm demonstrates robust performance on a wide variety of tasks: walking, swimming and hopping gaits. The performance, in terms of the total reward received, is similar to the DQN algorithm (Section 4-2) on the Atari games domain.

Generalized Advantage Estimation

Generalized Advantage Estimation (GAE) [6] builds on the TRPO and tackles the problem of having to require a large number of samples for policy improvement. The algorithm uses an exponentially weighted estimate of the advantage function, instead of the Q-function in (4-5), for reducing the variance of policy gradient estimates.

The algorithm is able to learn policies for highly challenging 3D locomotion tasks like running gaits for bipedal and quadrupedal robots but the training usually requires 1-2 weeks of real time data.

4-4-2 Guided Policy Search

Guided policy search (GPS) [4] provide another set of methods which have been used to learn neural network policies for high dimensional tasks. The method learns policies which map raw camera observations to joint motor torques. The method is also extremely sample efficient which becomes critical for tasks which involve real robots.

The main idea behind the algorithm is to first do a trajectory optimization for the given task and learn an optimal distribution over trajectories. The samples from the optimal distribution are used to train a neural network policy using supervised learning.

The trajectory optimization phase uses the full knowledge of the state of the system. This is a form of *instrumented training*, which greatly reduces the sample complexity of the trajectory optimization phase. To further reduce sample complexity, the layers of the neural network are first trained to predict certain elements of the state space which are not observable.

4-5 Comparison of DRL Methods

The DRL algorithms can be compared on 4 different criterion:

- Scalability: All DRL methods, apart from DQN, have shown that they could be applied on continuous high dimensional tasks (Table 1).
- Sample Efficiency: GPS is the most sample efficient since it utilizes trajectory optimization along with training the neural network. Policy gradient methods have been shown to be least sample efficient (Table 1).
- Robustness: TRPO, TRPO-GAE and GPS are more robust since they guarantee monotonic improvements in the policy with the optimization procedure. DDPG and DQN are less stable.

- Ability to learn from off policy data: DQN and DDPG can learn from off-policy data since they use a replay buffer. Policy gradient methods and GPS require on-policy data to train the network.

Table 1: Comparison of DRL methods based on sample efficiency. The number in the brackets denote the dimensionality of the system and these systems are taken from the original papers for each of the methods [6, 5, 3, 2, 4]. GPS does not provide number of iterations for either trajectory optimization or supervised learning.

	TRPO	**TRPO-GAE**	**DQN**	**DDPG**
Tested platforms (dimensions)	Atari Swimmer (10) Hopper (12) 2D Walker (20)	3D Biped (33) Quadruped (29)	Atari	MuJoCo (2-37)
Maximum iterations	200 million	50 million	50 million	2.5 million

Table 2: Comparison between TRPO and DDPG based on the average return for various tasks (MuJoCo environment) [52]

	TRPO	**DDPG**
Cart-Pole	4869	4634
Acrobat	-326	-223
Mountain Car	-61	-288
2D Walker	1353	318
Half-Cheetah	1914	2148
Full Humanoid	287	119

As seen from the above comparison, no algorithm satisfies all the criteria. For robotics, sample efficiency and the ability to learn from off-policy data are most crucial. Thus, DDPG or GPS are the promising options. But GPS requires a certain basic level of competence for the task to be learned - to initialize the parameters of the policy. This would mean designing walking controllers for the biped which is a challenging task in itself. Therefore, DDPG is the most promising choice because it is completely model-free and requires no knowledge of the environment, even though it might require slightly more samples for convergence.

Chapter 5

DDPG for Ideal Conditions

5-1 Introduction

The previous chapter introduced the Deep Deterministic Policy Gradient algorithm and showed how the algorithm compares to other DRL algorithms for high dimensional tasks. The algorithm shows good performance on high dimensional tasks despite using lesser number of iterations for convergence as compared to policy based algorithms. Also, the algorithm is completely model-free and requires no knowledge of the environment and has the ability to learn from off-policy data.

This chapter looks at the implementation of the DDPG algorithm for learning a walking task for robot LEO in a simulated environment. The results are compared with traditional RL algorithms like SARSA followed by a discussion on the performance.

5-2 LEO

LEO [16] is a 2D bipedal robot developed by the Delft BioRobotics Lab. The robot is attached to a boom which goes around in circles. In this thesis, a simulation environment for LEO is used. The model of the robot is taken from [17] where it was modeled using the rigid body dynamics simulator, Open Dynamics Engine (ODE). The boom is modeled by adding an extra mass at the hip. The actual robot and the model of the robot are shown in Figure 6.

LEO has 6 actuators, two each for the hips, knees and ankles. The state space of LEO comprises of 14 dimensions which consist of the angles and the angular velocities for the torso, hips, knees and ankles. The angles for all the joints of LEO are defined with respect to its parent body. The action space of LEO consist of voltages to be applied to each actuator.

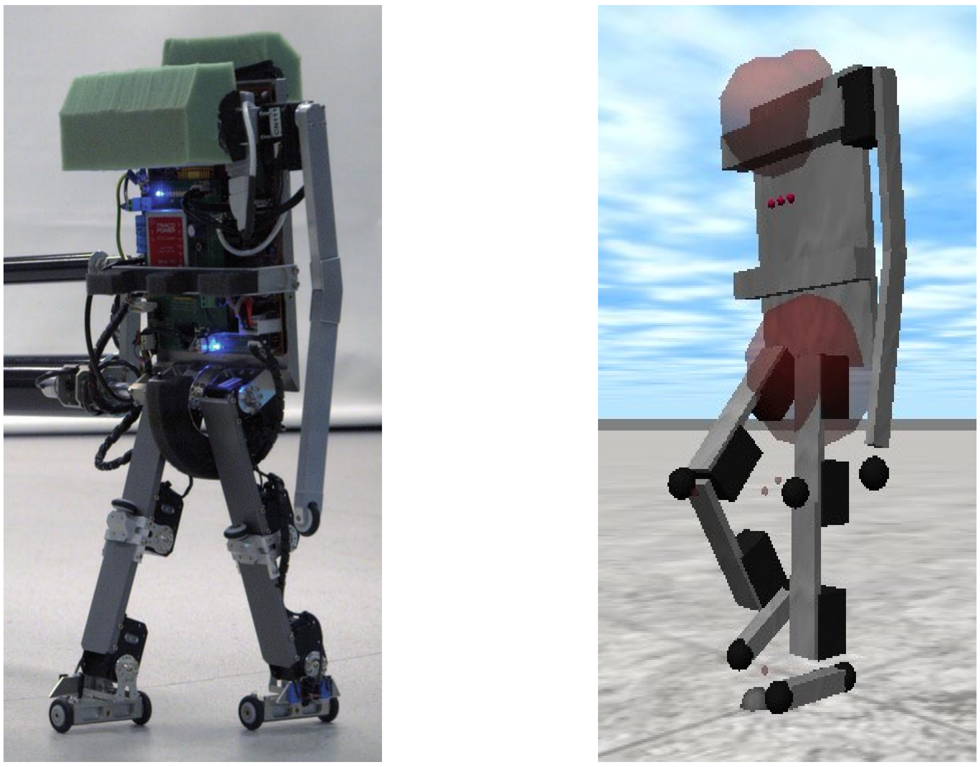


Figure 6: Left: Robot LEO. Right: Simulated model of LEO.

The state space and action space for LEO can be written as:

$$s = \begin{bmatrix} \phi_{\text{torso}} \\ \dot{\phi}_{\text{torso}} \\ \phi_{\text{l.hip}} \\ \dot{\phi}_{\text{l.hip}} \\ \phi_{\text{r.hip}} \\ \dot{\phi}_{\text{r.hip}} \\ \phi_{\text{l.knee}} \\ \dot{\phi}_{\text{l.knee}} \\ \phi_{\text{r.knee}} \\ \dot{\phi}_{\text{r.knee}} \\ \phi_{\text{l.ankle}} \\ \dot{\phi}_{\text{l.ankle}} \\ \phi_{\text{r.ankle}} \\ \dot{\phi}_{\text{r.ankle}} \end{bmatrix} \qquad a = \begin{bmatrix} U_{\text{l.hip}} \\ U_{\text{r.hip}} \\ U_{\text{l.knee}} \\ U_{\text{r.knee}} \\ U_{\text{l.ankle}} \\ U_{\text{r.ankle}} \end{bmatrix}$$

where subscript l and r denote the left and right leg respectively,
U is the voltage to be applied to each actuator.

Each episode of walking lasts for 25 s unless it is terminated prematurely. The termination condition of the episode is based on the angle constraints of the torso, the ankles of the robot and the height of the hip of the robot. The episode is considered terminated if the torso angle is greater than 1 rad, $|\phi_{\text{torso}}| > 1$ rad, the ankle angles are greater than 1.13

rad, $|\phi_{\text{l.ankle}}| > 1.13$ rad, $|\phi_{\text{r.ankle}}| > 1.13$ rad or if the hip height is less than 0.15 m. The termination condition for the hips and the torso denote the falling condition of the robot. The constraints on the ankles are added to make sure that the robot learns to keep the ankles almost perpendicular to the lower leg while walking to ensure a more human-like walking gait. This constraint was added after observing some initial learned gaits on LEO where LEO learned to walk on its toes without actually placing the complete feet on the ground.

At the start of every episode, the robot is initialized in the position as shown in Figure 6, with one of the legs completely stretched and the other leg is a slightly bend position. For a test run, the state vector at the beginning of the episode is initialized as:

$$s = \begin{bmatrix} \phi_{\text{torso}} \\ \dot{\phi}_{\text{torso}} \\ \phi_{\text{l.hip}} \\ \dot{\phi}_{\text{l.hip}} \\ \phi_{\text{r.hip}} \\ \dot{\phi}_{\text{r.hip}} \\ \phi_{\text{l.knee}} \\ \dot{\phi}_{\text{l.knee}} \\ \phi_{\text{r.knee}} \\ \dot{\phi}_{\text{r.knee}} \\ \phi_{\text{l.ankle}} \\ \dot{\phi}_{\text{l.ankle}} \\ \phi_{\text{r.ankle}} \\ \dot{\phi}_{\text{r.ankle}} \end{bmatrix} = \begin{bmatrix} -0.10\,\text{rad} \\ 0\,\text{rad}\,\text{s}^{-1} \\ 0.10\text{ rad} \\ 0\,\text{rad}\,\text{s}^{-1} \\ 0.82\text{ rad} \\ 0\,\text{rad}\,\text{s}^{-1} \\ 0\text{ rad} \\ 0\,\text{rad}\,\text{s}^{-1} \\ -1.27\text{ rad} \\ 0\,\text{rad}\,\text{s}^{-1} \\ 0\text{ rad} \\ 0\,\text{rad}\,\text{s}^{-1} \\ 0\text{ rad} \\ 0\,\text{rad}\,\text{s}^{-1} \end{bmatrix}$$

For training runs, the angles are perturbed randomly by a maximum of $\pm 5°$ to ensure that the learned policy is robust to slight changes in the starting configuration of the robot.

A reduced model of Leo is also taken from [17] which only actuates 3 joints (hips and the swing knee) and uses a pre-programmed PD controller for the other 3 joints. This model is also used to compare the results of DDPG with SARSA since SARSA does not work for a full state space model of LEO.

The reward function for the model awards the agent 300 m^{-1} for every meter of forward movement of the robot. The agent is penalized for time usage with -1.5 for every time step and -2 J^{-1} for every Joule of electrical work done. The agent also receives a negative reward of -75 at every premature termination of the episode.

5-3 Learning Parameters for DDPG

The inputs to the DDPG algorithm is the state space of LEO. The outputs of the policy are the voltage to be applied to all the actuators. The voltages are converted to torques and then applied to the actuators.

The parameters for the DDPG algorithm are taken from the original paper [3]. The actor and critic networks are initialized with two neural networks having two hidden layers with 400 and 300 hidden neurons respectively. The training algorithm is chosen to be ADAM and the learning rates are taken to be 0.0001 and 0.001 for the actor and critic respectively. A discount factor of $\gamma = 0.99$ and target update, $\tau = 0.001$ is used. The weights and biases of the hidden neurons are chosen from a uniform distribution $[-\frac{1}{\sqrt{f}}, \frac{1}{\sqrt{f}}]$, where f is the number of inputs to the layer. The weights and biases for the output layer are taken from a uniform distribution $[-3 \times 10^{-3}, 3 \times 10^{-3}]$ to ensure that the outputs at the start of training are close to zero.

The exploration noise is modeled as an Ornstein-Uhlenbeck process with parameters, $\sigma = 0.2$ and $\theta = 0.15$. The outputs of the policy are clipped to lie between the actuator limits after the addition of noise.

5-4 Results and Discussion

The following figures show the performance of DDPG for two different models of LEO: reduced state space (Sym) and full state space (Full). The reduced state space model (Sym) actuates 3 joints: the hips and the swing knee while the full state space model (Full) actuates all the 6 joints. DDPG is run for 300000 iterations ($\sim$ 9000 s) and the results are averaged out for 30 runs and plotted with 95% confidence interval.

Figure 7 shows the cumulative reward as a function of training time. As seen from the Figure, DDPG-Sym outperforms DDPG-Full by finding a policy which is almost twice as good. This is expected since a reduced state space makes finding a better policy easier. The high reward for the reduced state space can also be explained by the fact that the performance of a RL algorithm, in general, is very susceptible to the reward function and the reward function used here was designed for a reduced state space of Leo by [17] and may not be the most ideal for the full state space model.

The initial parts of the learning curve also show the difference between the two models. In the DDPG-Sym case, the agent initially learns to place the swing foot and remain stationary in its place since three of the actuators are controlled by a pre-defined controller. This is shown by the high negative reward which the agent receives due to a negative penalty on time. The agent can then slowly learn to move forward which brings an increase in reward and a higher walked distance. This is in contrast to the DDPG-Full case where the agent mostly falls down immediately for nearly 2000 s before it learns to place one foot properly and walk forward.

DDPG-Full also has a lower confidence interval because of a few training runs where the agent is unable to find a good policy at all. For the distance walked also, a similar trend is observed as shown in Figure 8.

To prove the effectiveness of DDPG over other classical RL approaches like SARSA, the results for SARSA for a reduced state space model are included in Table 3. The results of SARSA are courtesy of Ivan Koryakovskiy. SARSA is only run for the reduced model since SARSA does not work with the full state space of LEO. As seen from the Table, DDPG-Sym outperforms SARSA by a factor of almost 2.

From Table 3, it can be seen that the performance of DDPG-Full is almost at par with SARSA. An interesting thing to note is that DDPG-Full has a lower average reward than SARSA even though DDPG-Full has a higher walking distance. This is probably because three of the joints are auto actuated in the SARSA case, which means that those three joints are optimized to

Table 3: Comparison of results for different algorithms

Algorithm	**Reward**	**Distance Walked** (m)
DDPG-Sym	4573.34 ± 174.92	11 ± 0.3
SARSA-Sym	2790.72 ± 182.01	7.80 ± 0.39
DDPG-Full	2462.32 ± 554.14	7.92 ± 1.49

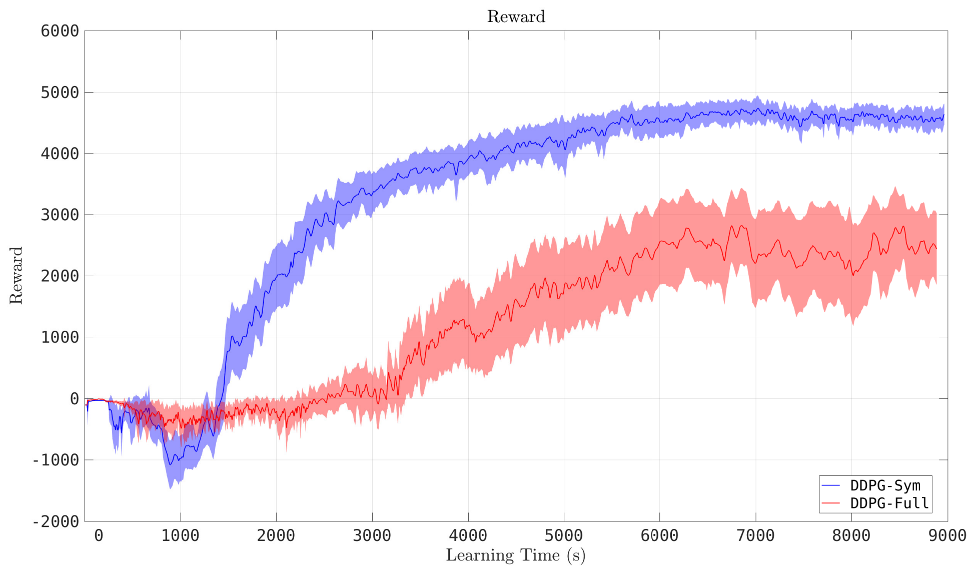


Figure 7: Performance curves for different models: DDPG for reduced model (blue), DDPG for full model (red).

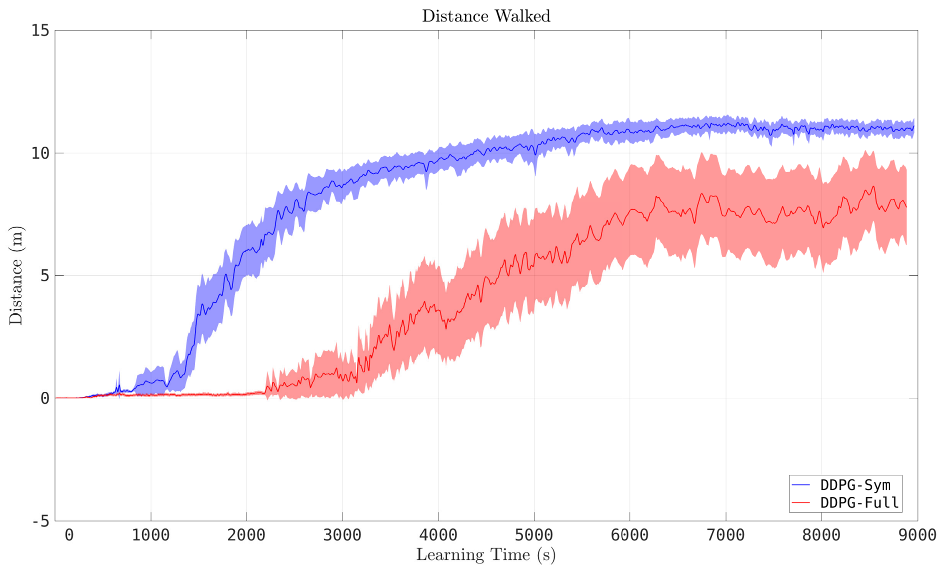


Figure 8: Distance walked by LEO for different models: DDPG for reduced model (blue), DDPG for full model (red).

move such that they use the least energy. This reduces the electrical work done by those joints and gives the agent lesser negative reward. For the DDPG-Full case, the agent has to learn how to move those joints which allows for greater change in those joints. This results in higher energy consumption which in turn gives the agent higher negative reward. But by having the liberty of changing the angles of those joints, the agent can find a policy which walks for a larger distance.

5-5 Summary

The chapter introduced the robot LEO and showed the results of implementing the DDPG for the robot. The DDPG algorithm robustly solves the walking problem of LEO for a continuous action space. Since, DDPG learns a policy for the complete state space of LEO, it means that no hand-coded controller is required to control certain joints of LEO, which is the case with SARSA. Even for a reduced state space model, DDPG outperforms SARSA by a significant margin. This clearly shows the advantage of applying DDPG for LEO as compared to other traditional RL algorithms like SARSA.

In spite of the performance, the main limitation of using the DDPG algorithm can be attributed to the number of samples and training episodes required to learn a policy. Those number of samples would be impossible to obtain for a real system in practice. Thus, a model learning approach will be introduced in the new chapter which will try to reduce the sample requirements from the real system while learning a policy which works optimally for the real system.

Chapter 6

Sample Efficient DDPG for Real World Conditions

6-1 Introduction

Model-based methods have been extensively studied for application of RL algorithms for robotic systems [33]. They are more favourable to model-free methods because they decrease the sample requirements from the real system considerably. Obtaining real data from robotic systems can be extremely difficult and time-consuming. For walking robots like LEO, obtaining data can be even more difficult since executing random policies on the robot could make the robot do extremely harmful maneuvers which could physically damage the robot. For DRL algorithms, which require something around 10^6 samples for their convergence, obtaining those number of samples on a real robot is impossible. Thus, a model learning algorithm is proposed which can learn a local difference model from running a few trials of a policy on the real system and then use the model in a DRL framework.

In this chapter, we will introduce our approach and motivate the choices made. We will show how a policy learned on the ideal model is updated to perform optimally on the real system, using a *difference model*, which is learned by taking samples from the real system.

6-2 Notation

- Ideal Model: Forward dynamics model based on a certain ideal configuration of the robot.

$$f_{\text{ideal}}(s_t, a_t) = s_{t+1}^{\text{ideal}}$$

- Real Model: Forward dynamics model based on the real configuration of the robot.

$$f_{\text{real}}(s_t, a_t) = s_{t+1}^{\text{real}}$$

- Difference Model: Dynamics model which predicts the difference in next states between the real and the ideal model given the same state and action.

$$d(s_t, a_t) = s_{t+1}^{\text{real}} - s_{t+1}^{\text{ideal}}$$

6-3 Proposed Method

A schematic of the proposed method is shown in Figure 9. The complete algorithm is shown in Algorithm 5. The main idea behind the approach is to learn a difference model which can capture the mismatch between the real system and the ideal model and then use the difference model to learn a new policy. The difference model is updated after learning every new policy. The algorithm starts by training an initial policy on the ideal model. This policy will generally behave sub-optimally when applied to the real system. The learned policy is then executed on the real system to obtain data from a few trajectories, $\{\tau_j\}_{j=1}^N$. A trajectory τ_j is a sequence of states and action: $\{s_1, a_1, s_2, a_2, \ldots\}$. While obtaining these trajectories, a certain amount of noise is added to the policy to ensure good exploration of the state space around the learned policy. The transitions obtained from the trajectories, $\{(s_{1:T}^{\text{real},j}, a_{1:T}^{\text{real},j}, s_{2:T+1}^{\text{real},j})\}_{j=1}^N$ is saved.

To obtain the same transition data for the ideal model, the $\{(s_{1:T}^{\text{real},j}, a_{1:T}^{\text{real},j})\}_{j=1}^N$ pairs is applied to the ideal model and next states, $\{s_{2:T+1}^{\text{ideal},j}\}_{j=1}^N$, calculated for each of the state-action pairs. The difference model is trained using the above data in a supervised learning manner. The architecture for the difference model is taken to be a Deep Neural Network (DNN). After learning a difference model, DDPG is used to learn a new policy with this difference model. The difference model compensates for the difference in states between the ideal model and the real system. Thus, the new policy obtained with the difference model will likely perform better on the real system. The entire process is iterative to enable the algorithm to converge to a good final policy which performs optimally with respect to the real system. The training of the new policy will have the actor and critic being bootstrapped from the previous policy and, thus, the DDPG algorithm will look to find an optimal policy in the neighborhood of the previous policy.

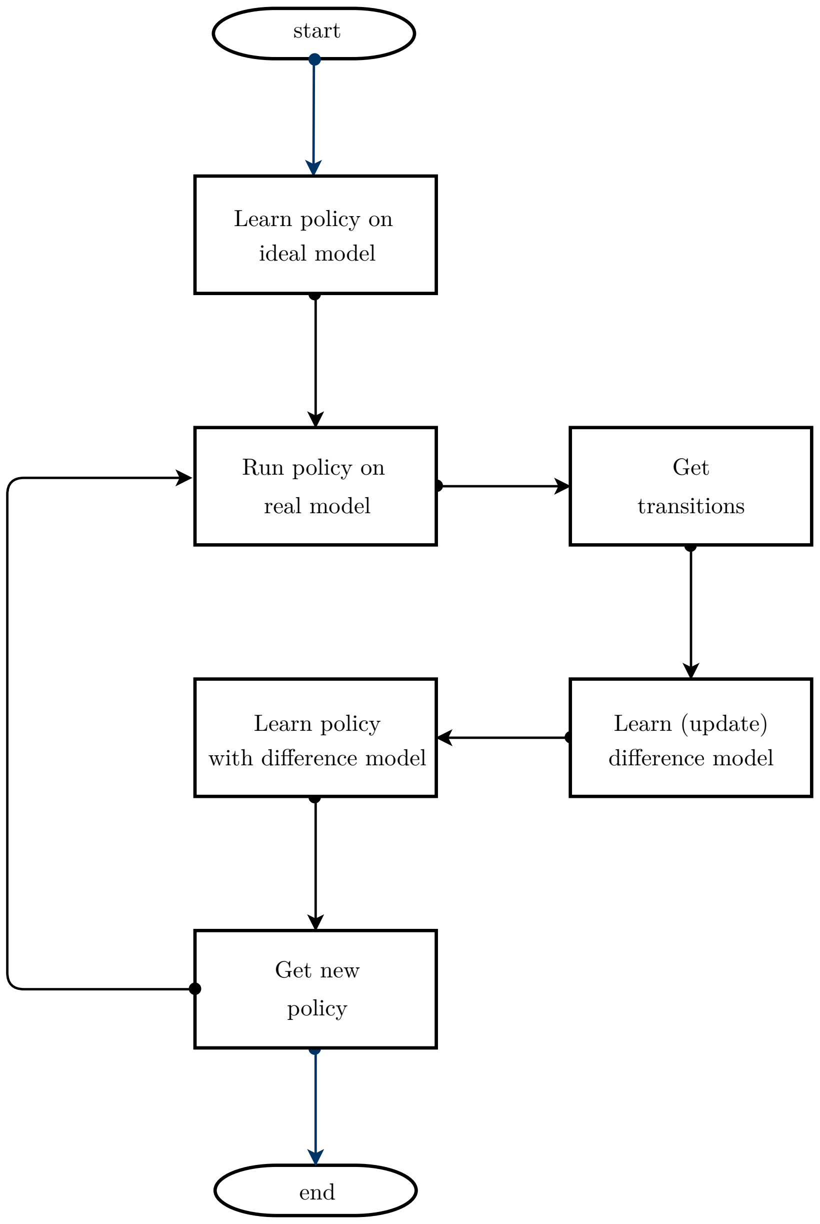

start
Learn policy on ideal model
Run policy on real model
Get transitions
Learn (update) difference model
Learn policy with difference model
Get new policy
end

Figure 9: Schematic of the proposed method

6-3-1 Algorithm

Algorithm 5 Proposed Algorithm

```
1: Initialize training data buffer $D$
2: Initialize critic network $Q$ and actor network $\mu$ with random weights $\theta^Q$ and $\theta^\mu$ respectively
3: Initialize target networks $Q'$ and $\mu'$ with weights $\theta^{Q'} \leftarrow \theta^Q,\ \theta^{\mu'} \leftarrow \theta^\mu$
4: Initialize a Deep Neural Network $d$
5: Learn a policy and Q-function using DDPG on the ideal model having final learned weights $\theta_0^\mu,\ \theta_0^Q$
6: for i=1,M do
7:     Initialize random process $\mathcal{N}$
8:     Execute policy $\mu(s;\theta_{i-1}^\mu) + \mathcal{N}$ on the real system to get transitions $(s_t^{\text{real}}, a_t^{\text{real}}, s_{t+1}^{\text{real}})$
9:     for each $(s_t^{\text{real}}, a_t^{\text{real}})$ pair do
10:        Calculate $s_{t+1}^{\text{ideal}} = f_{\text{ideal}}(s_t^{\text{real}}, a_t^{\text{real}})$
11:    end for
12:    Add training data to $D$
13:    Update $d$ using $D$
14:    Initialize actor and critic networks with weights $\theta_{i-1}^\mu,\ \theta_{i-1}^Q$
15:    Learn a new policy and Q-function having final learned weights $\theta_i^\mu,\ \theta_i^Q$ where
           $$s_{t+1} \rightarrow s_{t+1}^{\text{ideal}} + d(s_t, a_t)$$
16: end for
```

The individual parts of the algorithm are explained below:

Difference Model

The difference model, $d(s_t, a_t)$, is used to capture the mismatch in dynamics between the ideal model and the real system. The mismatch could come from model inaccuracies in mass, lengths of links, etc. and model uncertainties like friction, contact discontinuities, and measurement noise. The difference model should be able to accurately identify this mismatch for the data it is trained on and should also be able to generalize to other parts of the state space. The difference model considered in this method is a feed-forward DNN. The advantage of choosing a DNN is that it can be scaled up for high-dimensional systems and can learn complex representations from training data.

The DNN takes in as input the current state and action and predicts the difference in next states between the real system and the ideal model. The DNN is trained using stochastic gradient descent with samples obtained from the real system.

Training Data

The data used to train the difference model is obtained by running the previously obtained policy on the real system. Let the policy be denoted by $\mu(s;\theta_{i-1})$. The action to be taken is calculated as

$$a = \mu(s; \theta^{\mu}_{i-1}) + \mathcal{N}$$

where $\mathcal{N}$ denotes some random exploratory noise.

There is one advantage of collecting the transition data in this way. The distribution of data for learning the difference model should match the kind of states the real system is expected to observe when executing the policy. A good example to illustrate this point is that if the torso mass of the real system is higher than the torso mass of the ideal model, then the real system would tend to fall over if the original policy is applied on it. For the difference model to be effective, it should be accurate in those regions where the original policy has the most impact, due to the difference between the ideal model and the real system. This kind of transition data can only be obtained by running the policy on the real system and then evaluating the difference.

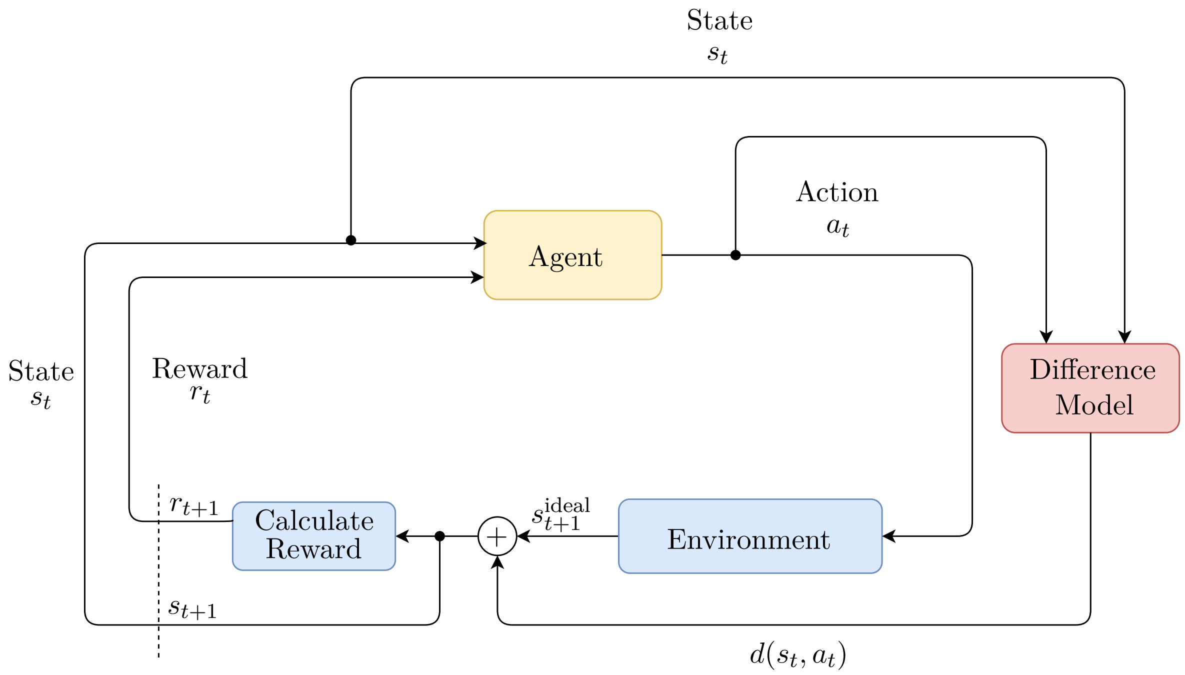


Figure 10: Integration of difference model with RL framework

Incorporating Difference Model with DDPG

With the difference model trained, DDPG is used to learn a new policy with the difference model included (Figure 10). The parameters of the actor and critic are bootstrapped from the previous policy.

For a state s_t, the action is given by the policy

$$a_t = \mu(s_t; \theta^{\mu}_{i}) + \mathcal{N}$$

The next state while applying the corresponding action is calculated as

$$s_{t+1} = s^{\text{ideal}}_{t+1} + d(s_t, a_t)$$

In the ideal scenario, if $d(s_t, a_t) = s_{t+1}^{\text{real}} - s_{t+1}^{\text{ideal}}$, then s_{t+1} would be the state corresponding to the real system. The new s_{t+1} is then used as the current state at the next time step. The rest of the DDPG algorithm works in a similar fashion as shown in Chapter 4.

6-4 Summary

The chapter introduces an iterative approach for learning an optimal policy for the real model using DDPG in a sample efficient way. A difference model is learned which can capture the 'simulation gap' between the ideal model and the real system. The difference model learned is a DNN. DDPG is then used to learn a new policy with the difference model included to offset the bias introduced due to the 'simulation gap'.

To test the effectiveness of the approach, two setups will be used in simulation. The first setup is a toy problem of an inverted pendulum and the second setup is LEO. The next chapter explains the results obtained by implementing the approach for the above two setups.

Chapter 7

Sample Efficient DDPG Results

7-1 Introduction

The last chapter introduced a sample efficient method for transferring policies learned on ideal models to real systems. In this chapter, we will test our approach using two simulation models. One of them would be referred to as the *ideal model* and the other would be referred to as the *real model*, which is an alternative to the real system. The chapter would look at the results of applying the algorithm for two experimental setups in simulation. The first system is a toy example of an inverted pendulum which will act as a proof of concept. The second system is LEO. The results for both the systems and some issues which arise from applying the above approach for LEO will also be discussed.

7-2 Experimental Setups

In this thesis, two experimental setups are used to evaluate the performance of the proposed approach, an inverted pendulum and the bipedal robot, LEO. The two setups are fundamentally different from each other in terms of dimensionality, stability and contacts.

7-2-1 Inverted Pendulum

The inverted pendulum, shown in Figure 11, is a low dimensional control problem. This makes the setup ideal for providing a proof of concept for the approach and also provide insights into the working of the algorithm. The pendulum has two states, angular position and angular velocity of the pendulum. The state space for the pendulum is augmented by replacing the angular position with the linear positions in x and y directions, which increases the state dimensionality to 3. The pendulum has one control input, the torque to be applied to the pendulum. The maximum torque is bounded to prevent the pendulum to do a swing-up in one go. The agent has to learn to swing the pendulum in one direction to gain momentum

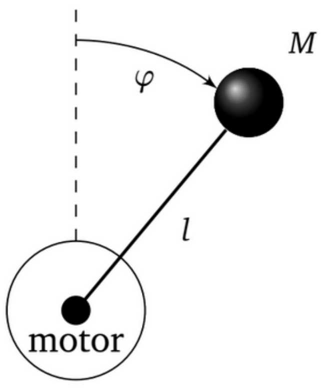


Figure 11: Inverted Pendulum model [53]

and then try swinging it up from the other side. The physical parameters for the pendulum in the ideal case are given in the table below:

Table 4: Pendulum Model Parameters [53]

Model parameter	**Symbol**	**Value**
Mass	m	0.055 kg
Inertia	J	0.000191 kg m^2
Length	l	0.042 m
Damping	b	0.000003 N m s/rad
Torque Constant	K	0.0536 N m/A
Rotor Resistance	R	9.50 Ω

The reward function for the pendulum is given by:

$$r = -5\varphi^2 - 0.1\dot{\varphi}^2 - 0.01a^2$$

where φ is the angular position of the pendulum, $\dot{\varphi}$ is the angular velocity of the pendulum and a is the control input.

The upright position of the pendulum is denoted by $\varphi = 0$. Thus, the agent is penalized for how far it is from the upright position. The agent also receives a negative reward for velocity and the amount of torque applied. The pendulum has no terminal state, and each episode comprises of 20 s. The pendulum is modeled using differential equations.

For the real model, the mass of the pendulum is increased to 0.09 kg to make sure that the policy learned on the ideal model fails to perform the task on the real model.

7-2-2 LEO

The simulator used for this part of the thesis for LEO is the Rigid Body Dynamics Library (RBDL) [54]. The simulator had to be changed from ODE since ODE did not have a functionality of providing a particular state to the actuators at any instant of time, which is extremely critical for our approach.

A floating base model for LEO is defined in RBDL which increases the state dimensionality of the system to 18, since the linear positions and velocities of the torso are also added to the state. The contact modelling for LEO assumes no slipping at the time of contact.

The reward function and the termination condition for the RBDL model is similar to the one used for the ODE model (Section 5-2).

The ideal model for LEO has the same configuration that was used in ODE. For the real model, the torso mass of the robot is increased to 1.25 kg to introduce instability and to ensure the original policy does not work optimally for the real model. Aside from the increase in mass, a certain viscous friction is also added to the motors as this is something which is encountered in the real LEO. The friction greatly effects the performance of the policy when applied on LEO, leading to oscillations and eventual fall of the robot. This value of friction is also extremely difficult to identify using standard parametric identification. For our case, the value of the friction coefficient is taken to be 0.03. The change in parameters are shown in Table 5.

Table 5: Difference in physical parameters between ideal and real model

	Torso Mass (in kg)	**Viscous Friction Coefficient**
Ideal Model	0.94226	0
Real Model	1.25	0.03

7-3 Learning Parameters

Training Data

The training data for learning the difference model is collected by running the previously obtained policy on the real model. Before each model update, 5000 data-points are collected and saved. The data-set is split into training and validation set in a 3:1 ratio, i.e., the training data-set has 3750 data-points, and the validation data-set has 1250 data-points. The number of training data collected per model update is the same for the inverted pendulum and LEO.

Difference Model

The DNN has three hidden layers with 400 neurons in each layer. The activation function for each of the hidden layers layer is taken to be *ReLu* while the output layer has a linear activation function. The input layer of the DNN has 24 inputs (18 states and 6 actions) for

LEO and 4 inputs (3 states and 1 action) for pendulum. Similarly, the output has 18 states for LEO and 3 states for pendulum.

To reduce over-fitting to seen data, *dropout* is used with a dropout probability of 0.3. The dropout is only applied to the hidden layers and taken to be the same for all the hidden layers. Dropout helps to improve the mean performance on the validation set and also makes the learning more stable as shown in Figure 12. Using dropout also reduces the need to choose an exact number of epochs for training, as would be required if the same accuracy is required without any dropout.

The DNN is trained by using a variant of stochastic gradient descent (SGD) method, AdaGrad. The advantage of using AdaGrad over normal stochastic gradient is the ability to adapt the learning rates based on the training data. This implies that the initial learning rate does not have much impact on the performance of the algorithm. AdaGrad is also recommended if the training data is sparse which is true for our case as well.

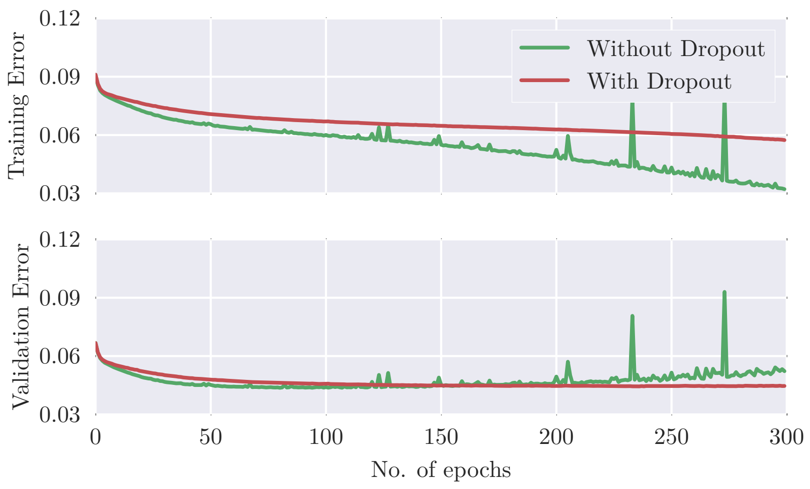


Figure 12: Training and test errors for training data with and without dropout (LEO).

DDPG with Difference Model

On the DDPG side, apart from the composition of the replay buffer and the noise parameters, the other things are kept constant as from Section 5-3.

Since DDPG is used to learn a new policy after every update of the difference model, the replay buffer from the first update of the policy with the difference model is saved and used in every following update. This helps to reduce the number of iterations required to find a good policy at each step.

The noise is modeled as an Ornstein-Uhlenbeck process with parameters $\sigma = 0.12$ and $\theta = 0.15$.

7-4 Results and Discussion

Figure 13 and Figure 16 show the performance of different policies on the real model for the inverted pendulum and LEO respectively. The cumulative reward is plotted against the model updates. For every model update, the performance of the best policy is considered. The horizontal lines denote benchmarks which can be used to compare how good a policy learned with a difference model is. The lower horizontal line denotes the performance of a policy which was learned on an ideal model of the system. The upper line denotes the best policy that was obtained by learning on a real model from scratch. This line is obviously not possible to get in practice, but it serves as a good baseline for rating the policies learned with the difference model. Ideally, when learning with a difference model, we would want the performance of a final policy to be as close to the top line as possible as that is the optimal policy for the model. [1]

Inverted Pendulum

As seen from Figure 13, the difference model is able to find an optimal policy in only one model update. This is expected since the dimensionality of the system is very small, so one update is enough to get a good difference model which can generalize well to different parts of the state space.

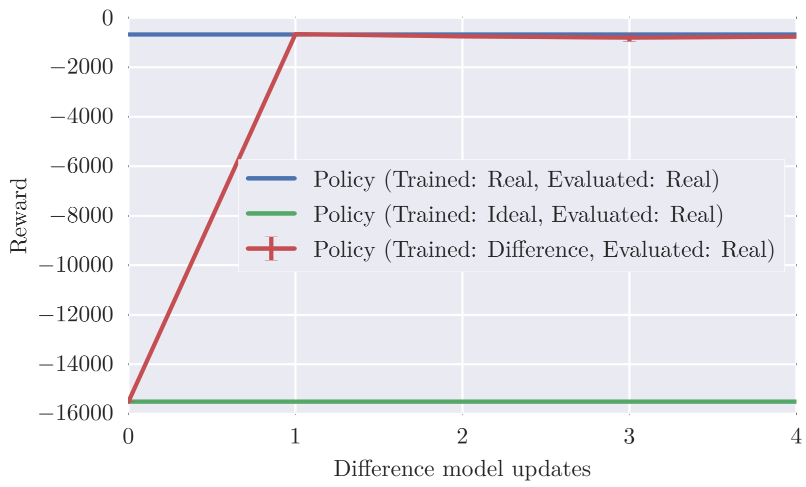


Figure 13: Comparison of results by evaluating the performance of different policies on the real model for the inverted pendulum. The *blue* line shows the performance of a policy which is directly trained on the real model from scratch. The *green* line shows the performance of a policy trained on the ideal model. The *red* lines show the mean of the rewards earned for policies trained with the difference model. The results with the difference model are averaged over 5 runs and plotted with 95% confidence interval.

[1]In the results, we have used three models for comparison: the ideal model, the real model, and the (ideal+difference) model. A policy can be learned and evaluated on any of the three models. The plots indicate on which model the policy has been trained on and on which model it is being evaluated. For brevity, the (ideal+difference) model is usually written as just the difference model.

To make sure that the policy found with the difference model is optimal when compared to the policy trained directly on the real model, the state trajectories and control input are compared for policies learned on different models and evaluated on the real model. Figure 14 shows that the distribution of states for the policies learned with the difference model and the real model is the same which shows that the difference model can capture the model differences accurately and help to learn a policy which performs optimally for the real model. The policy which is just trained on the ideal model fails to perform the task on the real model since it is unable to gather enough momentum to swing the pendulum up and keeps oscillating about the downward position.

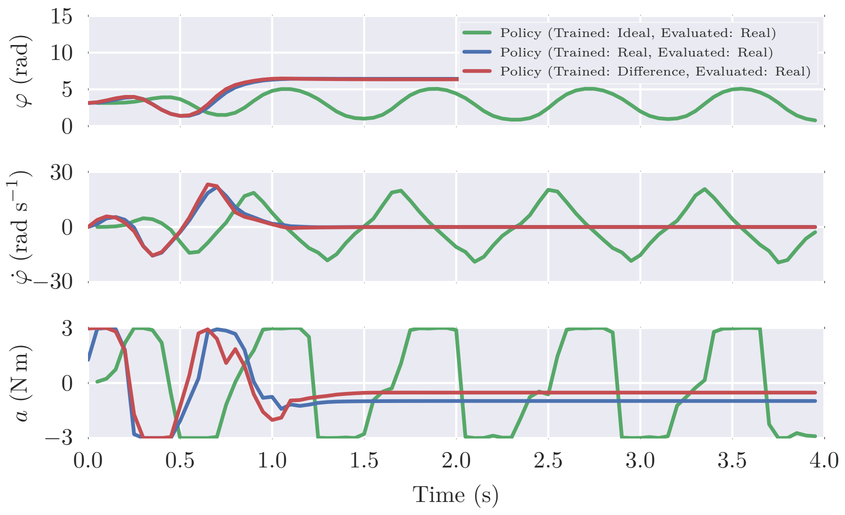


Figure 14: State trajectories and control input for policies learned on different models and evaluated on the real model. The distribution of states for the policies trained with the difference model and the real model are similar which shows that the difference model can compensate for the 'simulation bias' accurately.

The reason as to why the policy trained on just the ideal model does not work for the real model can also be shown by comparing the state distributions and control input for successfully learned policies for the ideal and real model (Figure 15). For the ideal model, the successful policy learns that it just needs to swing the pendulum in one direction to gain momentum and then it can complete the task. But for the real model, the successful policy requires two swings before it can gather enough momentum to swing the pendulum up. This can be seen by two crossings of $\varphi = \pi/2$ for the real model as compared to only one for the ideal case. This shows the effectiveness of the difference model in learning an optimal policy for the real model and provides a proof of concept of the developed approach.

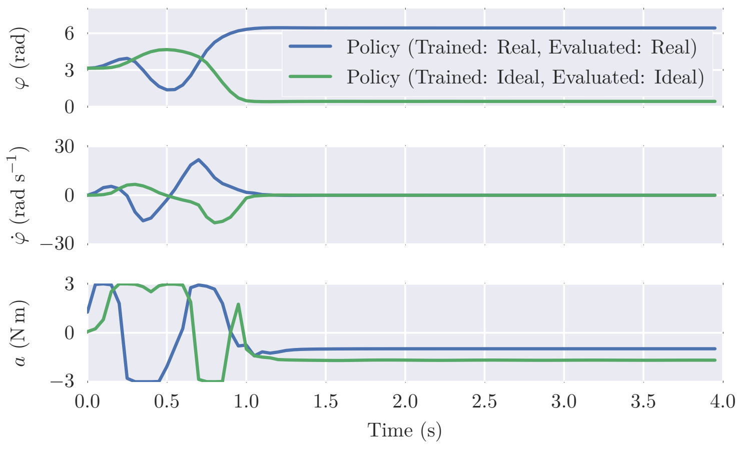


Figure 15: State trajectories and control input for successfully learned policies on the ideal and real model.

LEO

For LEO, Figure 16 shows that the difference model does provide an improvement to the policy, but it requires almost nine model updates before the policy stabilizes to a good final reward.

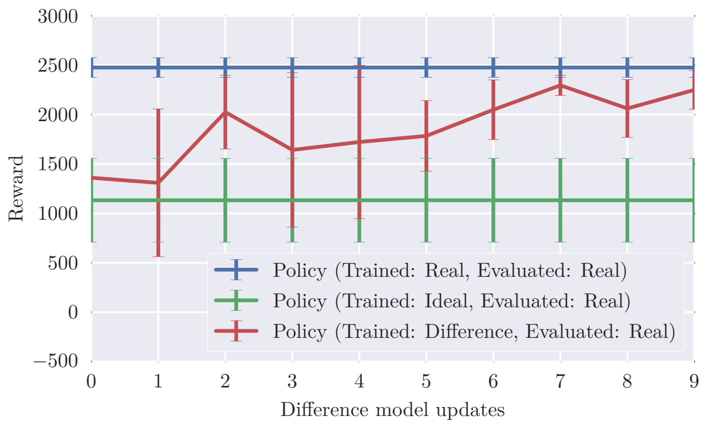


Figure 16: Comparison of results by evaluating the performance of different policies on the real model for LEO. The *blue* line shows the performance of a policy which is directly trained on the real model from scratch. The *green* line shows the performance of a policy trained on the ideal model. The *red* lines show the mean of the rewards earned for policies trained with the difference model. The results with the difference model are averaged over 5 runs and plotted with 95% confidence interval.

Table 6 compares the final performance of different learned policies evaluated on the real model in terms of final distance walked by the centre of mass (COM) of the torso and the electrical work done by the motors. The formula used for calculating the electrical work done by LEO is given by

$$E = \sum_j \frac{VI}{\nu}$$
$$I = \frac{V - k_d\omega}{R}$$

where V is the voltage to be applied to the motors, I is the current flowing though the motor, ω is the angular velocity of the motor, ν is the sampling frequency (30 Hz), R is the motor resistance, k_d is a constant and j is the number of actuators.

Table 6: Comparison of performance of different policies evaluated on the real model

	Distance Walked (m)	**Energy Consumption** ($\mathrm{J\,m^{-1}}$)
Policy (Trained: Ideal)	11.345	126.396
Policy (Trained: Difference)	14.791	110.434
Policy (Trained: Real)	15.481	90.540

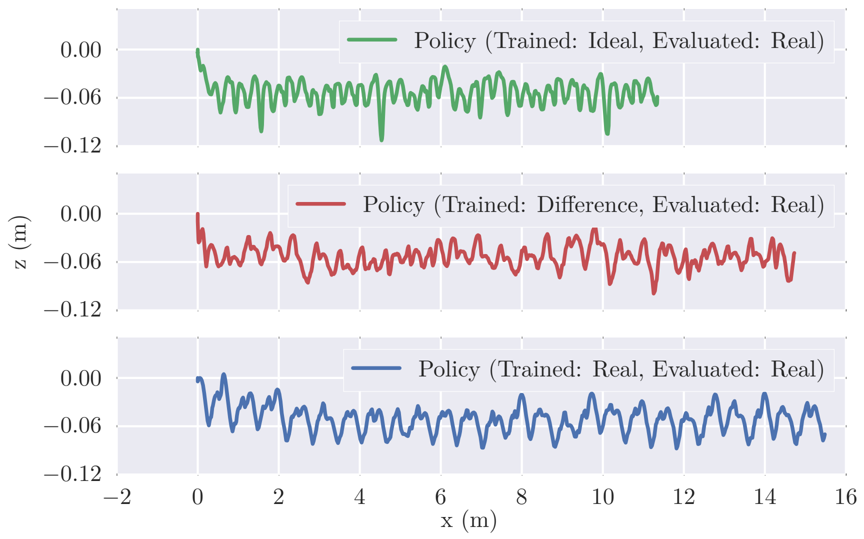


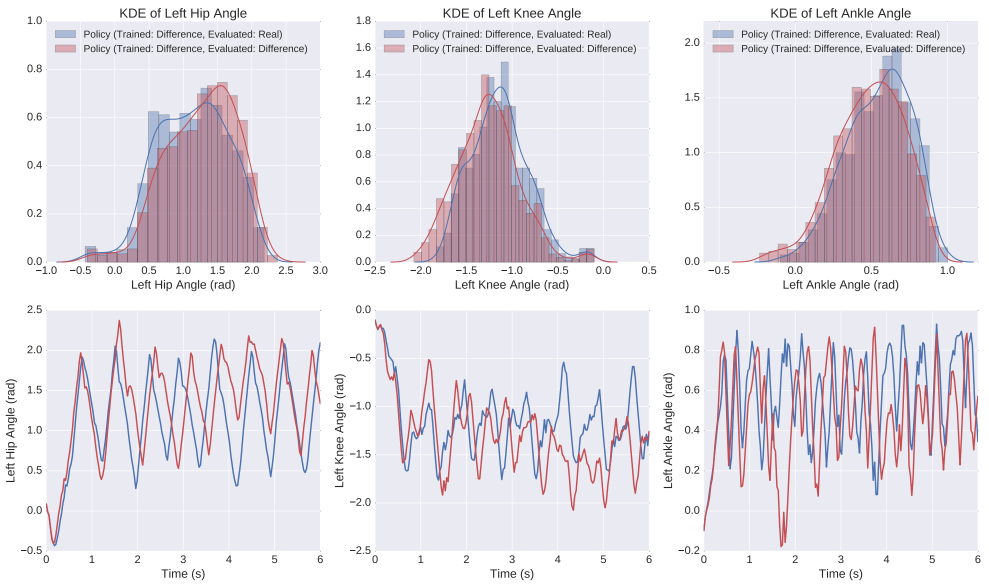
Figure 17: Trajectory of centre of mass (COM) of the torso for different policies evaluated on the real model. The x-axis indicate the distance walked by LEO controlled by each policy. Each walking trial lasted for 20 s.

It can be clearly seen from Table 6 that the policies learned with the difference model provide almost 30 % improvement in the distance walked and a 15 % reduction in energy consumption as compared to the policies trained only on the ideal model. Figure 17 also shows the trajectory of the COM of the torso for the different policies. The trajectory of the COM tells us about the kind of gait the robot learns since the COM of the robot can be assumed to be close to the torso. As expected, the ideal model policy performs the worst since its gait has a small step size and a tendency to lift the swing knee very high sometimes which reduces its walking speed. The difference model improves on the ideal model policy by reducing how high the swing knee goes. This results in a higher walking speed. This ensures that the distance walked by the difference model policy is within 10 % of the real model policy.

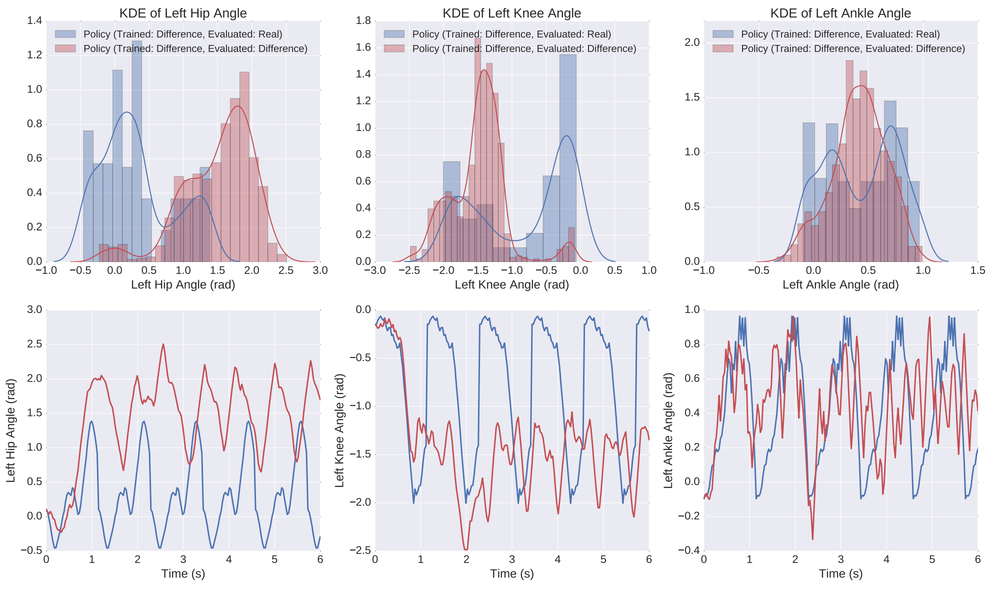
The learning curve of LEO (Figure 16) also shows a very high standard deviation for initial updates of the difference model. This implies that there is a higher chance for a policy trained with the difference model to fail on the real robot. A possible reason for this could be that the policy might explore regions of the state space where the difference model is not accurate enough. This can happen because there are no constraints on the policy update and the policy is free to wander into regions of the state space where the difference model has little or no data. This increases the chances of the DDPG algorithm to learn a policy with the difference model which fails to perform on the real model. The situation is shown in Figure 18b. As seen from Figure 18b, the distribution of states with the policy evaluated on the difference model and the real model are entirely different. This implies that the difference model makes a small error in prediction somewhere. Therefore, when the same policy is applied on the real model, the policy finds itself in a region where it was not supposed to be and fails. The problem is further compounded by the fact that in the real model the torso mass is higher, which reduces the region of stability for the robot. This implies that there is a very small difference between a successful and an unsuccessful policy. This can make a small error in the difference model catastrophic for the robot.

But, this behaviour is also beneficial for future updates since it allows for more data in regions where the difference model was not accurate earlier. Therefore, as the model updates, the difference model gets more samples in regions which it had little data before, thus, improving its prediction in such regions. This reduces the likelihood of the learned policy failing on the real model. This is seen in Figure 18a where the distribution of data for the difference model and the real model are roughly the same. Thus, the standard deviation of the reward reduces with the number of updates and the final policy learned is within 10 % of the optimal policy on the real model. [2]

[2]The videos for the policies learned on the inverted pendulum and LEO can be seen at:
`https://www.dropbox.com/s/e3md9lm82tb1ud5/Pendulum_policies.mp4?dl=0` (Pendulum),
`https://www.dropbox.com/s/2rrkw2vddpwi6yg/LEO_policies.mp4?dl=0` (LEO).



(a) Policy works for the real model



(b) Policy does not work for the real model

Figure 18: Distribution of states when executing a policy learned with the difference model on the real model and the difference model itself. The kernel density estimate (KDE) is plotted along with the propagation of states with time. The similarity in the distribution of states in the top figure indicate that the policy trained with the difference model does work on the real model as opposed to the bottom figure where the distributions are entirely different. This implies that in the bottom figure the policy deviates into a region of the state space where the difference model makes an error and that makes the learned policy fail on the real model.

Comparing Difference Models

Another thing to note from Figure 13 and Figure 16 is the fact that the difference model is able to find a policy which is optimal for the pendulum but not for LEO. To compare as to why the difference model for LEO is not able to find a policy which is very near to optimal, the quality of the difference model for the two setups is compared. The quality of the difference model can be compared by computing the mean squared errors on the validation data for both the setups. The formula used to compute the mean squared errors is given below:

$$MSE = \frac{1}{NP}\left|\sum_{k=1}^{N}\Big(f_{\text{real}}(s_k, a_k) - (f_{\text{ideal}}(s_k, a_k) + d(s_k, a_k))\Big)^2\right|_1$$

where N is the number of data points and P is the state dimensionality of the system.

Ideally, the difference model should push the error between the (ideal+difference) model and the real model to zero. But as seen from Table 7, this is not the case with LEO. For LEO, the reduction in error is only 41% which could be the reason as to why the policy learned with the difference model does not perform as well as learning on a real model from scratch.

Table 7: Reduction in error with difference model

	Mean error before training	Mean error after training	% Decrease
LEO	0.0763	0.0448	41.3
Inverted Pendulum	2.9638	0.0059	99.8

To investigate as to why the reduction in error is low in the case of LEO, the actual output of the training data for the two systems is plotted. Figure 19 shows the actual output data for one of the states for both the setups. It can be seen that in the case of the LEO, the actual data is more chaotic and has a higher frequency as compared to the pendulum. The chaotic nature of the data primarily come from the contact-rich dynamics in the case of LEO which adds discontinuity to the states. This high-frequency data makes it difficult for the DNN to accurately track the states and thus, it achieves low prediction accuracy on the data. The low prediction accuracy directly effects the quality of the policy that can be learned with the difference model and thus, the policy cannot match the performance obtained via learning on a real model.

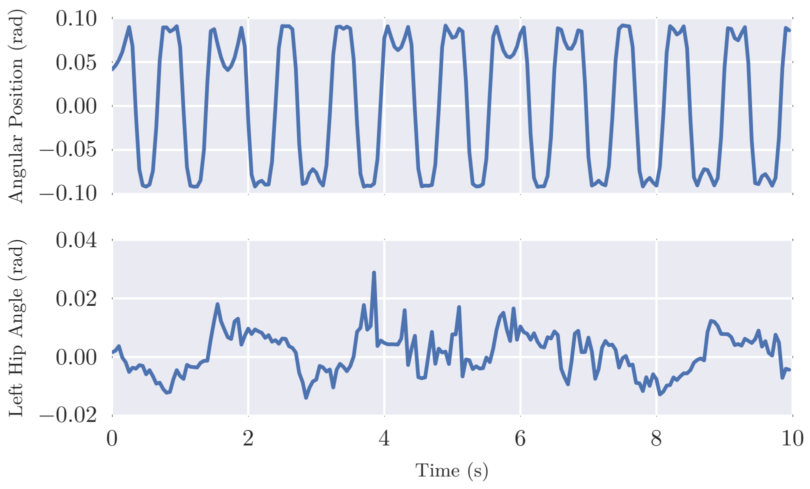


Figure 19: Training data for the difference model for pendulum (top) and LEO (bottom). Only one state is shown for both setups.

7-5 Advantages of the approach

The proposed approach provides a lot of advantages over model-free DDPG and other model based methods:

- **Works for high dimensional systems**: The approach is shown to work for LEO, which has a state dimensionality of 18 and an action dimensionality of 6. This can be considered as a very high dimensional system. The system also has a contact-rich dynamics and although, the final policy obtained with the difference model is not optimal, it provides good improvements to the final learned policy. Thus, this approach can be applied to a vast number of systems in practice.
- **Reduced interaction time required with the real system**: The approach greatly reduces the amount of interaction time required with the real system. Without the approach, it was seen that LEO required 150 min and pendulum required almost 30 min of real time data to learn an optimal policy. With this approach, the number of samples required from the real system are significantly reduced with only 3 min of real time data required for the pendulum and around 15 min of real time data required for LEO. This shows a 90 % reduction in real time data required.
- **Only learned policies executed on the real system**: The policy executed on the real system for getting the transition data have already been trained on the ideal model or with the difference model. This ensures that the actuation signals sent to the real system while executing such policies are not random and this reduces the amount of damage to the robot. On the contrary, if random initialized policies are applied on the real system, it would lead to a lot more damage and wear and tear on the robot.
- **No hand-coded controller needs to be designed**: Few model based approaches that have been previously used for bipedal walking use a pre-structured policy to reduce the number of real time interactions required with the system. This would mean

designing hand-coded controllers for the real system which are used to initialize the parameters of the policy. As already seen, this is an an extremely difficult task to do for bipedal walking problems. For our approach, we use DDPG itself to learn an initial policy which is beneficial since it requires no expert knowledge of the system to design controllers on the part of the programmer.

7-6 Limitations of the approach

Even though the difference model approach provides improvements in the policy learned, there are a few limitations of the approach as explained below:

- **No convergence guarantees**: As is most often the case with most algorithms that employ the use of neural networks, there are no convergence guarantees with the algorithm. The only way in practice for stopping the algorithm would be to execute the policy found with the difference model at every update on the real system and to stop the algorithm when the policy stops improving for a few updates.
- **Number of updates required**: The number of updates required by the difference model to find a good policy can vary greatly depending on the dimensionality of the problem. For the inverted pendulum, which is a low dimensional problem, the number of updates required is just one. Whereas, for LEO, which is a high dimensional system, the number of updates required are almost nine. Thus, there is no way of accurately predicting beforehand how many updates would be required for a particular problem.
- **Training the neural network at every iteration**: The training of the neural network is done in an iterative fashion. The neural network used is a simple feed-forward network without any memory. Thus, to ensure that the neural network does not forget what it learned during the previous updates, all the training data needs to be saved and the neural network is trained on the complete data-set every time. This increases the memory requirement at every update and might not be the most ideal approach in cases where more updates are required to learn an optimal final policy. Another issue which arises by training the neural network in this way is with dropout. With increasing data, one might want to change the dropout probability to ensure there is no over-fitting on the training data. Thus, the dropout probability also becomes a tuning parameter in cases where the number of updates increases.
- **Systems with contacts**: Systems which have contacts introduce non-differentiability in the propagation of states. This non-differentiability can introduce highly chaotic training data which could be difficult for the neural network to predict for high dimensional systems. This would directly impact the kind of policies that could be learned with the difference model which might lead to the approach not being completely effective. Also, if the difference becomes very large in this case, then also this approach may not work. This is something which is observed for the case for LEO with higher mass of the torso.

7-7 Summary

The chapter discusses the results of applying the proposed approach on two experimental setups, the inverted pendulum and LEO. It is shown that the policies learned with the difference model perform significantly better than learning on an ideal model only. The policy obtained for the inverted pendulum is optimal for real model, whereas, for LEO, even though the final policy obtained is not optimal, it does provide significant improvements. The number of samples required from the real system have also reduced with only 10 % of the samples required compared to when learning on a real model from scratch. Therefore, the proposed approach solves the problem of reducing the number of real time interactions required with the real system while transferring a policy learned on the ideal model optimally to the real system. Despite the advantages that this approach provides, it also has a few drawbacks but these drawbacks are similar to most approaches which use neural networks as function approximators or for learning a model.

Chapter 8

Conclusion

Deep Reinforcement Learning is the future of Reinforcement Learning. From achieving human like performance in games to be able to make a 3D humanoid walk from scratch, it has shown tremendous potential. Although, application of DRL to real robots is still limited. In this thesis, a DRL algorithm known as the Deep Deterministic Policy Gradient is used to learn a policy for LEO. LEO is 2D bipedal robot designed by the Delft BioRobotics Lab to be able to learn walking using learning methods like RL. Earlier it was shown that it was possible to learn a walking policy for LEO on the simulated environment but not on a real LEO from scratch. The real LEO used to break down after every 5 minutes of training which made it difficult to apply any RL algorithm on it. Real LEO finally learned to walk when showed with some demonstrations of walking using a hand-coded controller. Also, the RL algorithm only worked if only 3 joints were actuated for LEO as compared to the complete 6 joints. The other 3 joints were controlled using a pre-programmed controller.

The main aim of applying DDPG for LEO was to see if a policy could be learned to control LEO for the complete action space. In Chapter 5, it was shown that DDPG was able to learn a policy for the full LEO. This was the first time that a RL algorithm was used to learn a policy for the full state space model of LEO. Also, to compare how DDPG fared with regards to traditional RL algorithms like SARSA, DDPG was also run for a reduced state space for LEO. It was seen that DDPG outperformed SARSA by a factor of 2.

Even though DDPG was able to learn a policy for LEO, it required 3×10^5 samples for convergence. This would mean 150 min of training in real time. So this approach would be infeasible to learn a policy for the real LEO. Also, the naive approach of learning a policy on the simulated model and applying it on real LEO also does not work because there are always some discrepancies between the model and the real system which makes the policy learned on the simulated model behave sub-optimally when applied on the real system.

To overcome this issue, Chapter 6 introduced an iterative approach of learning a difference model which can capture the mismatch between the simulated model and the real system, also known as the 'simulation gap'. The difference model will learn to predict the difference in next states between the two systems. The difference model will be trained by obtaining the transitions data by executing certain policies on the real LEO. With the difference model learned, DDPG will be used to learn a new policy with the difference model included to offset the bias introduced by the 'simulation gap'. The difference model will make sure that the new policy learned will likely perform optimally on the real system. The approach serves the dual purpose of being generic and also reducing the sample requirements from the real system.

The approach is tested in Chapter 7 on two experimental setups, the inverted pendulum and LEO. The approach is tested in simulation by defining two models, the ideal model and the real model. The ideal model is based on certain ideal configuration of the robot and the real model is made by increasing masses in the ideal model and adding friction in the case of LEO. The difference model is able to find a policy which performs significantly better on the real system as compared to a policy only trained on the ideal model. For pendulum, the difference model is able to find a policy which is optimal whereas, for LEO, the policy found with the difference model is within 10 % of the optimal policy. This clearly shows the effectiveness of using the DNN as a model architecture for the difference model. Along with the increase in performance, the number of samples required from the real system has significantly reduced with only 3 min of real time data needed for the pendulum and around 15 min (1/10th) of real time data needed for LEO.

In summary, it can be seen that the approach is shown to work for two systems which are very diverse. Along with the success of the algorithm, the approach also has a few limitations. The algorithm provides no convergence guarantees, as is often the case with most algorithms which use neural networks. The number of difference model updates required for the algorithm to converge are difficult to predict beforehand for a system. The training of the neural network for the difference model would require saving all the transition data which can be very memory intensive if many updates are required for a system. In spite of these limitations, the approach does seem like a promising way to learn control policies for real systems using DRL algorithms.

Directions for Future Work

The thesis provided interesting results in the field of application of DRL algorithms for real systems. The following could be possible improvements or future research directions:

- The DDPG algorithm implemented with the difference model has no constraints on the updates of the policy. This leads to high variation in rewards during the first few updates of the model since the policy might deviate into regions where the difference model has less samples and is not accurate. This can be prevented by having constraints on the updates of the policy which would prevent the policy from deviating too far from the previous learned policy. This might lead to the algorithm requiring more updates of the difference model and larger training times but might lead to more stable learning. Constrained policy optimization for actor critic methods have been introduced in [55] by introducing a risk factor in the reward. This would a possible direction for research in case of such problems.

- The difference model learned in this approach is a DNN which does not provide any confidence bounds about the prediction it makes. In order to provide some confidence about the prediction, the approach proposed by [56] was investigated but it did not provide any good results. So, it would be good to extend the model of the difference model to a Bayesian DNN which can provide certain uncertainty bounds on the prediction. This uncertainty could be added to the reward as an additional cost which will try to restrict the policy from exploring such regions of the state space.

- The thesis experimented with two setups in simulation to prove the effectiveness of the proposed approach. Therefore, the next step would be to try to apply it on a real system and see if it does work the same way as is does in simulation. Additionally, two setups are generally not enough to test out certain aspects of the algorithm such as the number of updates that would be required for a particular problem. Thus, more experimentation is required with systems with different dimensionality to accurately figure out for what problems this approach performs the best and this might also provide rough estimates on the number of updates that might be required for a particular problem.
- It was shown that the difference model was not able to find an optimal policy for LEO with the difference model. A reason for it was attributed to the chaotic nature of the actual output of the training data which made it difficult for the DNN to accurately track the states. On the other hand, if a full dynamic model is learned, then the training data will be a lot smoother and it will be easier for a DNN to accurately track the states. But it will require a lot more samples to learn an accurate model. Thus, a trade-off exists in this case. Hence, more experimentation with systems having contact-rich dynamics is required to figure out which model type is more ideal.

Bibliography

[1] P. S. Rodman and H. M. McHenry, "Bioenergetics and the origin of hominid bipedalism," *American Journal of Physical Anthropology*, vol. 52, pp. 103–106, Jan 1980.

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–533, Feb 2015.

[3] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, pp. 1–14, 2015.

[4] S. Levine, C. Finn, T. Darrell, and P. Abbeel, "End-to-End Training of Deep Visuomotor Policies," *Journal of Machine Learning Research*, vol. 17, pp. 1–40, 2016.

[5] J. Schulman, S. Levine, M. Jordan, and P. Abbeel, "Trust Region Policy Optimization," *ICML-2015*, p. 16, 2015.

[6] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, "High-Dimensional Continuous Control Using Generalized Advantage Estimation," *arXiv*, pp. 1–9, 2016.

[7] M. Vukobratovic, B. Borovac, D. Surla, and D. Stokic, *Biped locomotion: dynamics, stability, control and application*, vol. 7. Springer Berlin Heidelberg, 2012.

[8] S. Kajita, F. Kanehiro, K. Kaneko, K. Fujiwara, K. Harada, K. Yokoi, and H. Hirukawa, "Biped walking pattern generation by using preview control of zero-moment point," in *IEEE International Conference on Robotics and Automation, 2003. Proceedings. ICRA '03.*, vol. 2, pp. 1620–1626, IEEE, 2003.

[9] T. McGeer, "Passive Dynamic Walking," *The International Journal of Robotics Research*, vol. 9, pp. 62–82, Apr 1990.

[10] S. Collins, A. Ruina, R. Tedrake, and M. Wisse, "Efficient bipedal robots based on passive-dynamic walkers.," *Science (New York, N.Y.)*, vol. 307, no. 5712, pp. 1082–5, 2005.

[11] R. Tedrake, I. R. Manchester, M. Tobenkin, and J. W. Roberts, "LQR-trees: Feedback Motion Planning via Sums-of-Squares Verification," *The International Journal of Robotics Research*, vol. 29, pp. 1038–1052, Jul 2010.

[12] A. Herdt, H. Diedam, P.-B. Wieber, D. Dimitrov, K. Mombaur, and M. Diehl, "Online Walking Motion Generation with Automatic Footstep Placement," *Advanced Robotics*, vol. 24, pp. 719–737, Jan 2010.

[13] J. W. Grizzle, C. Chevallereau, R. W. Sinnet, and A. D. Ames, "Models, feedback control, and open problems of 3D bipedal robotic walking," *Automatica*, vol. 50, pp. 1955–1988, Aug 2014.

[14] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction.* Cambridge, MA: MIT Press, 1 ed., 1998.

[15] J. Kober, J. A. Bagnell, and J. Peters, "Reinforcement learning in robotics: A survey," *The International Journal of Robotics Research*, vol. 32, no. 11, pp. 1238–1274, 2013.

[16] E. Schuitema, M. Wisse, T. Ramakers, and P. Jonker, "The design of LEO: A 2D bipedal walking robot for online autonomous reinforcement learning," in *IEEE/RSJ 2010 International Conference on Intelligent Robots and Systems, IROS 2010 - Conference Proceedings*, pp. 3238–3243, IEEE, Oct 2010.

[17] E. Schuitema, *Reinforcement learning on autonomous humanoid robots.* PhD thesis, 2012.

[18] B. Vennemann, "Sample-Efficient Reinforcement Learning for Walking Robots," 2013.

[19] J. Morimoto, G. Cheng, C. G. Atkeson, and G. Zeglin, "A Simple Reinforcement Learning Algorithm For Biped Walking," *International Conference on Robotics and Automation*, pp. 3030–3035, April 2004.

[20] T. Mori, Y. Nakamura, M. A. Sato, and S. Ishii, "Reinforcement Learning for a CPG-driven Biped Robot," *AAAI 2004*, pp. 623–630, 2004.

[21] T. Matsubara, J. Morimoto, J. Nakanishi, M. A. Sato, and K. Doya, "Learning CPG-based biped locomotion with a policy gradient method," *Proceedings of 2005 5th IEEE-RAS International Conference on Humanoid Robots*, vol. 2005, pp. 208–213, 2005.

[22] R. Tedrake, T. W. Zhang, and H. S. Seung, "Stochastic Policy Gradient Reinforcement Learning on a Simple 3D Biped," *IEEE/RSJ International Conference on Intelligent Robots and Systems*, vol. 3, pp. 2849–2854, 2004.

[23] E. Schuitema, D. Hobbelen, P. Jonker, M. Wisse, and J. Karssen, "Using a controller based on reinforcement learning for a passive dynamic walking robot," *5th IEEE-RAS International Conference on Humanoid Robots*, pp. 232–237, 2005.

[24] G. Cheng, “Learning CPG-based Biped Locomotion with a Policy Gradient Method: Application to a Humanoid Robot,” *The International Journal of Robotics Research*, vol. 27, pp. 213–228, Feb 2008.

[25] T. Hester, M. Quinlan, and P. Stone, “RTMBA: A real-time model-based reinforcement learning architecture for robot control,” in *Proceedings - IEEE International Conference on Robotics and Automation*, pp. 85–90, IEEE, May 2012.

[26] M. P. Deisenroth and C. E. Rasmussen, “PILCO: A Model-Based and Data-Efficient Approach to Policy Search,” *Proceedings of the International Conference on Machine Learning*, pp. 465–472, 2011.

[27] S. Ha and K. Yamane, “Reducing hardware experiments for model learning and policy optimization,” in *Proceedings - IEEE International Conference on Robotics and Automation*, vol. 2015-June, pp. 2620–2626, IEEE, May 2015.

[28] P. Christiano, Z. Shah, I. Mordatch, J. Schneider, T. Blackwell, J. Tobin, P. Abbeel, and W. Zaremba, “Transfer from Simulation to Real World through Learning Deep Inverse Dynamics Model,” *arXiv preprint*, Oct 2016.

[29] A. Punjani and P. Abbeel, “Deep learning helicopter dynamics models,” in *Proceedings - IEEE International Conference on Robotics and Automation*, vol. 2015-June, pp. 3223–3230, IEEE, May 2015.

[30] C. J. C. H. Watkins, *Learning from delayed rewards.* PhD thesis, King’s College, London, 1989.

[31] M. P. Deisenroth, G. Neumann, and J. Peters, “A Survey on Policy Search for Robotics,” *Foundations and Trends in Robotics*, vol. 2, no. 1, pp. 1–141, 2013.

[32] D. Silver, “Lecture notes on reinforcement learning,” February 2015.

[33] J. Kober and J. Peters, “Policy Search for Motor Primitives in Robotics,” *Machine Learning*, vol. 84, pp. 1–8, Jul 2011.

[34] T. Degris, M. White, and R. S. Sutton, “Off-policy actor-critic,” *CoRR*, vol. abs/1205.4839, 2012.

[35] V. R. Konda and J. N. Tsitsiklis, “On Actor-Critic Algorithms,” *SIAM Journal on Control and Optimization*, vol. 42, pp. 1143–1166, Jan 2003.

[36] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour, “Policy Gradient Methods for Reinforcement Learning with Function Approximation,” in *Advances in Neural Information Processing Systems*, pp. 1057–1063, 1999.

[37] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, “Deterministic Policy Gradient Algorithms,” *Proceedings of the 31st International Conference on Machine Learning (ICML-14)*, pp. 387–395, 2014.

[38] B. Costa, W. Caarls, and D. S. Menasché, “Dyna-MLAC: Trading computational and sample complexities in actor-critic reinforcement learning,” in *Proceedings - 2015 Brazilian Conference on Intelligent Systems, BRACIS 2015*, pp. 37–42, IEEE, Nov 2016.

[39] W. Caarls and E. Schuitema, "Parallel Online Temporal Difference Learning for Motor Control," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 27, pp. 1–12, Jul 2015.

[40] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba, "End to End Learning for Self-Driving Cars," *arXiv:1604*, pp. 1–9, Apr 2016.

[41] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," *International Conference on Learning Representations (ICRL)*, pp. 1–14, Sep 2015.

[42] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, and B. Kingsbury, "Deep Neural Networks for Acoustic Modeling in Speech Recognition," *IEEE Signal Processing Magazine*, vol. 29, pp. 82–97, Nov 2012.

[43] A. Jain, "Fundamentals of deep learning." `https://www.analyticsvidhya.com/blog/2016/03/introduction-deep-learning-fundamentals-neural-networks/`, 2016. Accessed: 2017-06-16.

[44] "Multilayer feedforward neural networks." `http://cse22-iiith.vlabs.ac.in/exp4/index.html`. Accessed: 2017-06-30.

[45] D. P. Kingma and J. L. Ba, "Adam: a Method for Stochastic Optimization," *International Conference on Learning Representations 2015*, pp. 1–15, Dec 2015.

[46] J. Duchi, E. Hazan, and Y. Singer, "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization," *Journal of Machine Learning Research*, vol. 12, pp. 2121–2159, Jul 2011.

[47] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*, vol. 15. MIT Press, 2014.

[48] S. Gu, T. Lillicrap, I. Sutskever, S. Levine, and S. Com, "Continuous Deep Q-Learning with Model-based Acceleration," *ICML*, Mar 2016.

[49] Z. Wang, V. Bapst, N. Heess, V. Mnih, R. Munos, K. Kavukcuoglu, and N. de Freitas, "Sample Efficient Actor-Critic with Experience Replay," *arXiv*, pp. 1–20, Nov 2016.

[50] E. Todorov, T. Erez, and Y. Tassa, "MuJoCo: A physics engine for model-based control," in *IEEE International Conference on Intelligent Robots and Systems*, pp. 5026–5033, 2012.

[51] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The arcade learning environment: An evaluation platform for general agents," in *IJCAI International Joint Conference on Artificial Intelligence*, pp. 4148–4152, Jul 2015.

[52] Y. Duan, X. Chen, J. Schulman, and P. Abbeel, "Benchmarking Deep Reinforcement Learning for Continuous Control," *arXiv*, vol. 48, p. 14, 2016.

[53] I. Grondman, *Online model learning algorithms for actor-critic control*. PhD thesis, 2015.

[54] M. L. Felis, "RBDL: an efficient rigid-body dynamics library using recursive algorithms," *Autonomous Robots*, vol. 41, pp. 495–511, Feb 2017.

[55] L. A. Prashanth and M. Ghavamzadeh, "Variance-constrained actor-critic algorithms for discounted and average reward MDPs," *Machine Learning*, vol. 105, pp. 367–417, Mar 2016.

[56] Y. Gal and Z. Ghahramani, "Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning," in *Proceedings of The 33rd International Conference on Machine Learning* (M. F. Balcan and K. Q. Weinberger, eds.), vol. 48 of *Proceedings of Machine Learning Research*, (New York, New York, USA), pp. 1050–1059, PMLR, 2016.