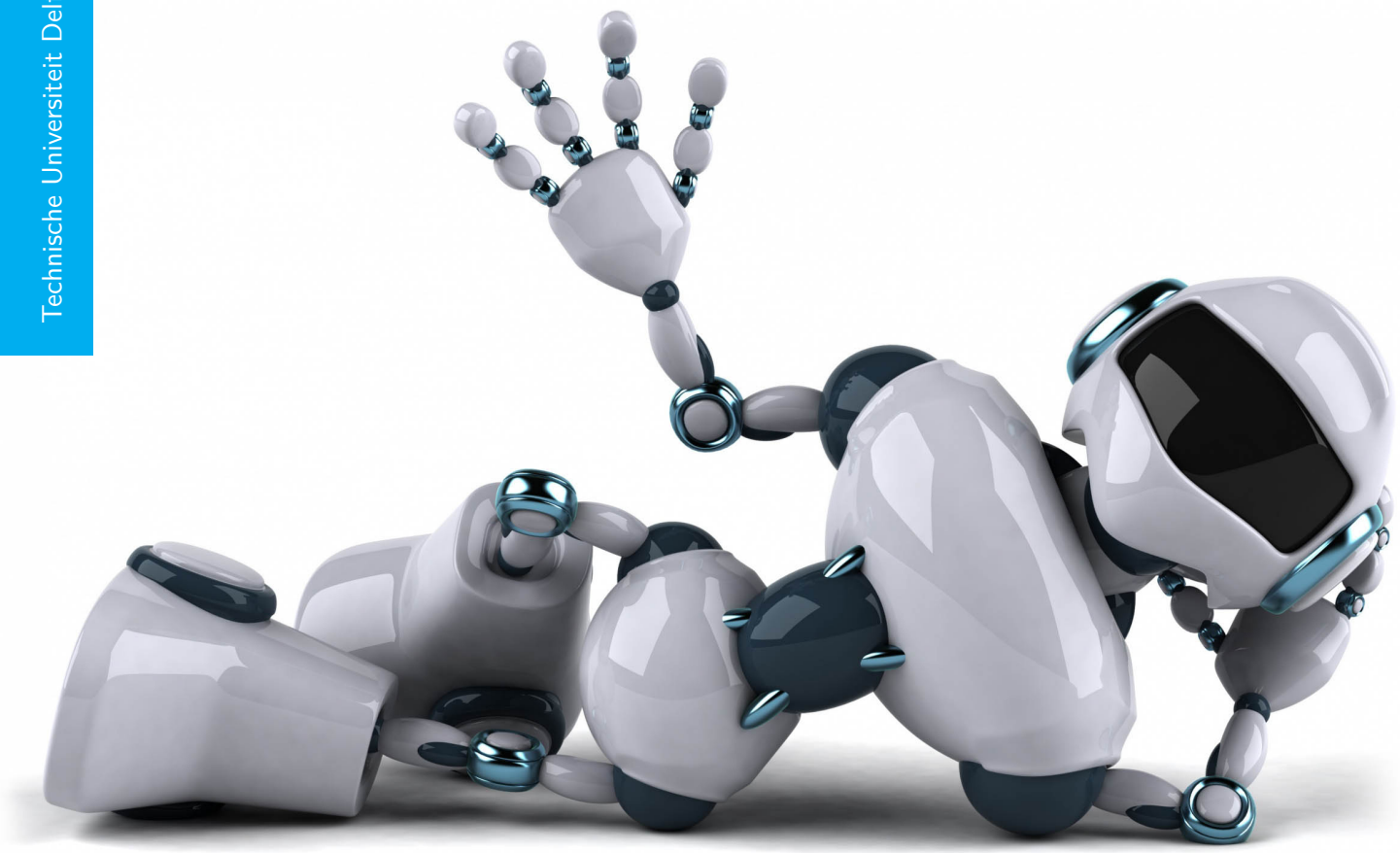
Deep Learning for Actor-Critic Reinforcement Learning

Xueli Jia

Technische Universiteit Delft



TU Delft Delft University of Technology

Challenge the future

DEEP LEARNING FOR ACTOR-CRITIC REINFORCEMENT LEARNING

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

Xueli Jia

Supervisor: Prof. dr. Robert Babuska

May 2015

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

TU Delft Delft University of Technology

ABSTRACT

Reinforcement Learning (RL) is a framework to deal with decision-making problems with the goal of finding optimal control policies. Robotics decision-making problems are of importance and provide interesting applications for reinforcement learning; however, since real-world robots have many degrees of freedom, high-dimensionality arises when applying reinforcement learning algorithms.

Deep Neural Networks (DNNs) have the potential to reduce the high-dimensional state spaces without losing information. A low-dimensional representation is generated by training a multilayer neural network with a small central layer to reconstruct high-dimensional input vectors. Back-propagation algorithms can be used to tune the weightings in this neural network with the help of the gradient descent method.

The objective of this project is to develop approaches of applying deep neural networks with reinforcement learning, hereby, efficiently generating dynamic policies in robot motion-control applications. More specifically, the objective of this project is to overcome the high-dimensionality challenge in reinforcement learning with the help of deep neural networks, a stacked auto-encoder, and using various types of sensory inputs like angles and angular velocities in the manipulator.

Experimentally, a simple segway system, which consists of a link and a wheel, is used to test performance. The main approach to improve the learning performance is to develop a low-dimensional representation replacing the original inputs that describe the current states of the segway system in the learning process and using a stacked auto-encoder instead.

Xueli Jia

Delft, July 2015

CONTENTS

ACKNOWLEDGEMENT

I dedicate this thesis to all who have helped with its completion: my supervisor, family, colleagues and friends.

First, I would like to thank my supervisor, Prof. Dr. Robert Babuška, for the opportunity of this challenging project and his detailed help on many professional questions, project management and academic writing. I have learned a lot from the meetings, the discussions and the feedback. I would also like to thank Ivo Grondman, Ivan Koryakovskiy, Wenjie Pei and Yu Hu for their profitable advice and Yudha Prawira for his help on experimental problems. In addition, I am grateful to my families for their support during my project completing tasks seemingly impossible.

Xueli Jia

Delft University of Technology

May, 2015

1

INTRODUCTION

Automatically acquiring a control policy is used in control problems with complex systems and environments that are difficult or impossible to model by classical approaches. It has been applied more and more for controlling robots in recent years. The control policy may be complicated despite the ease of the settings. For instance, a chess player typically loses when too many pieces are lost, or when checkmate appears unavoidable. However, to obtain a winning strategy for chess is very hard (computationally intractable). Those individuals who perform actions in such a game are usually called *agents* and the settings of such games as the *environments*.

For an agent, one attempt to win such games is to learn optimal control policies while interacting with its environment. As a result, the agent knows the best action to perform in a specific situation. Reinforcement Learning is one of such attempts to discover the optimal control. It was inspired by the psychology of human and animal behaviours. By sequentially giving a reward or a punishment to an individual after each action, the individual would conclude a strategy guiding its actions. For an agent, Reinforcement Learning is a computational approach of discovering an optimal strategies during such trial-and-error processes to maximize agent's reward while interacting with its environment (Sutton and Barto, 1998).

More recently, many challenges have arisen from applying reinforcement learning, especially problems with high-dimensional state spaces and continuous states and actions. Approaches to reduce the complexity of high-dimensional state spaces without limiting the capabilities of robots are desired. Deep neural network is a famous framework in machine learning and provides a possible

solution, especially for the generation of low-dimensional pixel spaces for digital images (Hinton and Salakhutdinov, 2006). This project aims at dimensional reduction in reinforcement learning using some deep neural networks with robot control applications.

This thesis is organised as follows: chapter 2 presents basics of reinforcement learning and neural networks, as well as the specific algorithms applied below. Following that, chapter 3 presents the implementation of a segway system using actor-critic reinforcement learning and states the challenge of high-dimensionality, as well as the solving method for the challenge. Then, chapter 4 shows the results for a well-trained stacked auto-encoder by collecting training data from the segway system. Finally, to reduce the dimension of the such problems, chapter 5 proposes a new approach by applying a stacked auto-encoder into reinforcement learning followed by analysis and evaluation.

2

BACKGROUND

This chapter provides basic knowledge of Reinforcement Learning (RL) and Deep Neural Networks (DNNs). More specifically, Section 2.1 provides the basics of reinforcement learning and the framework of the Markov decision process (MDP). Section 2.2 introduces the neural network of perceptrons and the unit structure of stacked auto-encoder.

2.1. REINFORCEMENT LEARNING

Reinforcement learning is inspired by human and animal behaviour whereby an individuals optimize their behaviours while interacting with their environments (Sutton and Barto, 1998). It is aimed at dealing with decision-making problems in automatic control and Artificial Intelligence (AI). The controllers, decision makers and individuals interacting with the system are referred as the *agents*. For the system is the *environment*. The reactions of agents are known as *behaviours*. The agent interacts with its environment by carrying out actions and receiving feedback. The behaviour of an agent is optimized through maximizing its value function, which is used to evaluate the quality of the decision of actions.

2.1.1. REINFORCEMENT LEARNING ELEMENTS

A framework with agents and an environment consists of four main elements: a reward function, a value function, a policy, and a reinforcement learning system (Sutton and Barto, 1998).

Reward Function At the heart of a reinforcement learning, problem lays the *reward function*. It defines the desirability of a receiving state (or state-action) and judges each event by a single value. The objective of the agent is to select a sequence of actions that maximize the total reward value. The simplest way to achieve this, is to use the greedy method which compares the reward for each candidate action in each state and selects the action sequence with the highest reward.

Value Function A *value function* defines long-term desirability. The value of a state is an expect accumulate reward for this state and all the followed states in the future. Different from a reward, which represents an immediate effect, a value is a long-term effect. In some situations, the reward for the current state is low, but may result in a high value eventually (or vice versa). The action(s) leading to the highest value are preferred since it gives a refined judgment for entire environment.

Policy A *policy* is a function to map the perceived states of the environment and dictates the actions to be taken when in those states. The policy is the core of a reinforcement learning agent, in the sense that it alone is sufficient to determine behavior.

Reinforcement Learning System A *reinforcement learning system* is a model of the environment that has ability of mimicking the behavior of the system. According to this model, the next state and reward might be predicted by given a state and action. In addition, a model is used for planning, it has ability of deciding a course of actions by considering possible future situations before they are actually experienced.

2.1.2. REINFORCEMENT LEARNING FRAMEWORK

Reinforcement learning enables an agent to deal with decision-making problems for an extended period of time. For each time step, the control performance is measured by a reward function. For the extended period, the control goal is to maximize the accumulated reward which is called *return*. An overview of reinforcement learning schema is shown in Figure 2.1. In such schema, an agent is assigned to a current state $s \in S$ and a reward signal $r \in R$ and then according to these data outputs an action $u \in A$ (where S is the set of all possible states, R is the set of all the rewards and A indicates the set of actions). The environment is mapped from a state s_k to a new state s_{k+1} given an action u_k: $s_{k+1} = f(s_k, u_k)$ (Arsén, 1990).

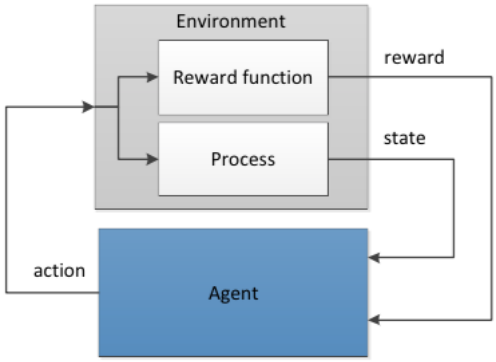


Figure 2.1: *The interaction between the agent and the environment in reinforcement learning system.*

MARKOV DECISION PROCESSES

A reinforcement learning model can be described as a Markov decision process (MDP). A MDP is defined as a 4-tuple $M = (S, A, f, \rho)$, where S indicates the state space, A denotes the action space, $f : S \times A \times S \to [0, \infty)$ is the state transition function, and $\rho : S \times A \times S \to R$ denotes the reward function. MDPs are tools for modeling sequential decision-making problems. For an agent in the current time step $k \in N$, let $s_k \in S$ and $u_k \in A$ denote the system state and the selected action respectively. When the action is applied by the system, the state is transferred from s_k to s_{k+1}:

$$s_{k+1} = f(s_k, u_k) \tag{2.1.1}$$

A reward r_{k+1} at time $k+1 \in N$ is:

$$r_{k+1} = \rho(s_k, u_k) \tag{2.1.2}$$

Both the next state s_{k+1} and the reward r_{k+1} are observed by the agent, who then performs a new action u_{k+1} for the system. A control goal is achieved by repeating this process.

The return defines the total discounted sum of the rewards received:

$$R = \sum_{k=0}^{\infty} \gamma^t r_{k+1} \tag{2.1.3}$$

Where $0 \leq \gamma < 1$ is a discount factor. It makes rewards in the future worth less than the immediate reward. This is the so-called discounted reward MDP. In contrast, discounted MDP has $\gamma = 1$. The result of MDP for an agent is a sequence of behaviors that maximize the expected return such that an optimal value is obtained.

2.1.3. REINFORCEMENT LEARNING METHODS

Reinforcement learning solving methods can be roughly divided into three groups (Grondman *et al.*, 2012a): critic-only, actor-only, and actor-critic methods. Critic-only methods select actions based

on the value function. While actor-only methods search for optimal polices in policy spaces. Actor-critic methods are combinations of critic-only and actor-only methods in order to extract their advantages and overcome their challenges.

CRITIC-ONLY METHODS

Critic-only methods generate optimal value functions first and then according to these function derive optimal policies. A value function is learned by solving the Bellman equation recursively. Q-learning (Bradtke *et al.*, 1994, Watkins and Dayan, 1992) and SARSA (Rummery and Niranjan, 1994) are two examples of critic-only methods. They estimate state-action value functions without explicit policy functions. A policy π is generated by solving the following optimization problem:

$$\pi(x) = argmax_u Q(x,u) \tag{2.1.4}$$

Where Q is the state-action value function, u indicates an action, and x is the system state.

Temporal Difference Methods Critic-only methods use Temporal Difference (TD) learning, resulting in a low variance in the estimates of expected returns (Sutton, 1988). TD error δ is defined as the error between the expected return and the received return:

$$\delta_{k+1} = \gamma V(s_{k+1}) - V(s_k) + r_{k+1} \tag{2.1.5}$$

Then the value function for a policy can be computed incrementally:

$$V_{k+1}(s_k) = V_k(s_k) + \alpha_k \delta_{k+1} \tag{2.1.6}$$

The parameter $\alpha_k \in (0,1]$ indicates the learning rate at time k, it is used to weight the value function. For continuous state and action spaces, it is clear that critic-only methods discretize the continuous actions and result in an enumerate optimization action space. Thus, the disadvantage of these methods is that they undermine the original continuous actions and the true optimum.

ACTOR-ONLY METHODS

Different from critic-only methods, actor-only methods search for the immediate optimal policy at each step by using policy gradient method. Typically, the class of policy is parameterized by some parameters $\vartheta \in R^p$, with p is the number of features used in the approximation, then the parameterized policy is indicated as π_ϑ (Grondman *et al.*, 2012a). Assuming the cost-to-go function J is differentiable with respect to the policy and the policy is differentiable with respect to parameter vector ϑ, the immediate cost function gradient can be calculated by:

$$\nabla_\vartheta J = \frac{\partial J}{\partial \pi_\vartheta} \frac{\partial \pi_\vartheta}{\partial \vartheta} \tag{2.1.7}$$

According to above function, a locally optimal solution can be found at each time step by using an optimization technique. Then the parameters ϑ are updated along the gradient.

$$\vartheta_{k+1} = \vartheta_k + \alpha_k \nabla_\vartheta J_k \tag{2.1.8}$$

where $\alpha_k > 0$ should small enough such that $J(\pi(\theta_k)) \leqslant J(\pi(\theta_{k+1}))$. Many methods are proposed to estimate the gradient, like infinitesimal perturbation analysis and likelihood-ratio methods introduced by Aleksandrov *et al.* (1968) and Glynn (1987), respectively.

Actor-only methods are popular because of their strong convergence property that inherit from the gradient descent approach. However, the disadvantages of these methods are the large variance of the estimated gradient (Sutton *et al.*, 1999) and the missing of knowledge about past estimations (Peters *et al.*, 2010).

ACTOR-CRITIC METHODS

Actor-critic methods combine the actor-only and critic-only methods aiming to gain both their advantages (Witten, 1977). Actor-only methods have ability of computing continuous actions without optimization procedures on value function. Critic-only methods estimate the expected return with a low variance and speed up the learning process. In actor-critic methods, the policy and the value function are updated separately. The actor represents the policy and the critic evaluates the current policy that prescribed by the actor. Actor-critic methods aim to find a parameter vector that maximizes the return by using a gradient updating approach (Grondman *et al.*, 2012a).

Figure 2.2 represents a structure of actor-critic algorithm. The intelligent agent is split into an actor and a critic. The learning process starts by giving an input u to the system and a reward is produced at the same time. Then the critic evaluates the policy for each immediate reward r. According to the critic information, actor generates a new input for the system. When this process is repeated, the actor and the critic are updated step by step until it achieves its control goal.

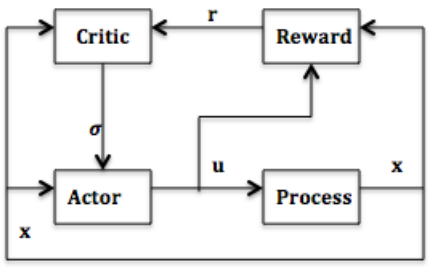


Figure 2.2: *Block diagram representation of actor-critic algorithm*

Control with Function Approximator The most simple way to represent a value function is by a table with one entry for each state. However, this method is limited with a small numbers of states and actions. For tasks with continuous or large number of states and actions, a function approximation is required to represent both the value functions and policies.

Unified notations for actor-critic algorithms generated by Sutton and Barto (1998) are used in this thesis. Sutton and Barto (1998) provides the basic temporal-difference actor-critic methods for updating rules. Using $V(x,\theta)$ denotes the approximate value function parameterized by θ, and ϕ indicates the features (basis function), the linear parameterization of the value function is described as:

$$V_\theta(x) = \theta^T \phi(x) \tag{2.1.9}$$

Temporal difference is defined as the predicted performance correction and it claims that a better performance is obtained than the predicted one if it receives a positive temporal difference. The temporal difference error δ is defined as:

$$\delta_k = r_k + \gamma V(x_k, \theta_{k-1}) - V(x_{k-1}, \theta_{k-1}) \tag{2.1.10}$$

Given the temporal difference and a learning rate $\alpha_c > 0$, the critic parameter vector θ can be updated using the gradient-descent rule:

$$\theta_k = \theta_{k-1} + \alpha_c \delta_k \frac{\partial V(x,\theta)}{\partial \theta}|_{x=x_{k-1},\theta=\theta_{k-1}} \tag{2.1.11}$$

A one-step backup problem is caused when updating the critic by applying (2.1.11), it results in a number of steps reward received in practice. Eligibility traces is a method used to solve this problem. It is implemented by assigning credit to states several steps earlier. For example, at time k and state x, the eligibility trace is represented as:

$$e_k(x) = \begin{cases} 1, & \text{if } x = x_k \\ \lambda\gamma e_{k-1}(x) & \text{otherwise} \end{cases} \tag{2.1.12}$$

where $0 \leq \lambda < 1$ is the trace decay rate, and γ indicates the decays with time. Taking this term into consideration, the update of θ can then be denoted as:

$$\theta_k = \theta_{k-1} + \alpha_c \delta_k \sum_{x_v \in X_v} \frac{\partial V(x,\theta)}{\partial \theta}|_{x=x_v,\theta=\theta_{k-1}} e_k(x_v) \tag{2.1.13}$$

where, X_v indicates all the states visited during the current trial.

Parameter ϑ parameterizes the approximate policy $\pi(x,\vartheta)$ and results in:

$$\pi_\vartheta(x) = \vartheta^T \phi(x) \tag{2.1.14}$$

To find an optimal result, reinforcement learning algorithms apply exploration to keep trying new actions. This can be achieved by involving a term named zero-mean random exploration, it is indicated as Δu_k

$$u_k = \pi(x_k, \vartheta_{k-1}) + \Delta u_k \tag{2.1.15}$$

Then the update rule for the actor is defined as:

$$\vartheta_k = \vartheta_{k-1} + \alpha_a \delta_k \Delta u_{k-1} \frac{\partial \pi(x, \vartheta)}{\partial \vartheta}|_{x=x_{k-1}, \vartheta=\vartheta_{k-1}} \tag{2.1.16}$$

where $\alpha_a > 0$ is the learning rate.

Different function approximators $\phi(x)$, such as radial basis functions (RBFs), local linear regression (LLR), and tile coding, can be used to test the performance of actor-critic algorithms. Both RBFs and LLR are continuously differentiable, so they can provide the gradient information. However, tile coding is suited for use on sequential digital computers and for efficient on-line learning because of its binary features. In this thesis, RBFs approximator is applied.

Radial Basis Functions A typical RBF feature i has a Gaussian response $\Phi_s(i)$ based on the distance between the current state x, and the center state c_i (Sutton and Barto, 1998)

$$\phi_s(i) = \exp(-\frac{||x - c_i||^2}{2\sigma_i^2}) \tag{2.1.17}$$

Where σ_i denotes the feature's width.

In this thesis, RBFs are used to approximate both actor and critic with linear combinations:

$$V_\theta(x) = \theta^T \phi(x) \tag{2.1.18}$$

$$\pi_\vartheta(x) = \vartheta^T \phi(x) \tag{2.1.19}$$

with $\phi(x)$ is a column of normalised RBFs $\phi_i(x)$:

$$\phi_i(x) = \frac{\tilde{\phi}_i(x)}{\sum_j \tilde{\phi}_j(x)} \tag{2.1.20}$$

Using a diagonal matrix B indicates the widths of RBFs, $\tilde{\phi}_i(x)$ is defined as:

$$\tilde{\phi}_i(x) = e^{-\frac{1}{2}(x-c_i)^T B^{-1}(x-c_i)} \tag{2.1.21}$$

Then the standard actor-critic algorithm can be implemented as Algorithm 1 (Grondman *et al.*, 2012b).

Algorithm 1 Actor-critic algorithm

Require: : the trace decay rate λ, the discount factor γ, critic learning rate α_c, and actor learning rate α_a.

1: initialize $e_0 = 0, \forall x$, and initialize θ_0, and ϑ_0.

2: Apply random input u_0

3: Start from $k \leftarrow 1$

4: **loop**

5: randomly choose Δu_k

6: Measure x_k, r_k

7: $u_k \leftarrow \pi(x_k, \vartheta_{k-1}) + \Delta u_k$

8: Apply u_k

9: $\delta_k = r_k + \gamma V(x_k, \theta_{k-1}) - V(x_{k-1}, \theta_{k-1})$

10: $e_k(x) = \begin{cases} 1, & \text{if } x = x_k \\ \lambda\gamma e_{k-1}(x) & \text{otherwise} \end{cases}$

11: $\theta_k = \theta_{k-1} + \alpha_c \delta_k \sum_{x_v \in X_v} \frac{\partial V(x,\theta)}{\partial \theta}|_{x=x_v, \theta=\theta_{k-1}} e_k(x_v)$

12: $\vartheta_k = \vartheta_{k-1} + \alpha_a \delta_k \Delta u_{k-1} \frac{\partial \pi(x,\vartheta)}{\partial \vartheta}|_{x=x_{k-1}, \vartheta=\vartheta_{k-1}}$

13: update k as $k \leftarrow k+1$

14: **end loop**

2.2. DEEP NEURAL NETWORK

Deep Learning is a new approach in Machine Learning. Ever since its development in 2006, it has been applied to diverse domains such as image, video, audio and text processing. Reducing the dimensionality of data using deep neural network techniques is popular when dealing with large quantities of video or images. Through training a multi-layer neural network with a small central layer to reconstruct high-dimensional input vectors, high-dimensional data can be can be converted to low-dimensional codes (Hinton and Salakhutdinov, 2006). In this section, two shallow Artificial Neural Networks (ANNs) perceptron and auto-encoder are introduced. Following that, Multi-Layer Perceptron (MLP) and Stacked Auto-Encoder are described in detail.

2.2.1. ARTIFICIAL NEURAL NETWORK

Artificial neural networks, invented by McCulloch and Pitts (1943), have become a popular and efficient way for solving complex computing problems. The framework was first created based on the structure of biological neurons and it was claimed that an artificial neuron has the capability of solving problems on the strength of a computational model called threshold logic (Hu, 1965). Layer-units in the model are taken into account as neurons that have abilities of learning new knowledge after training them with prior knowledge.

Perceptron, defined by Rosenblatt (1958), is the most simplest neural network framework. A model of perceptron is shown in Figure 2.3 (Nakanishi). It indicates that this framework only include two layers: an input layer and an output layer. The input layer accepts a number of n inputs, which can be used to generate a binary output to answer 'yes' or 'no' or '0' or '1' questions. According to this graph, the data-processing procedure can be summarized as two steps; first, the input values are put together to generate a weight summation, and then the generated summation is dealt with by an activation function to produce an output.

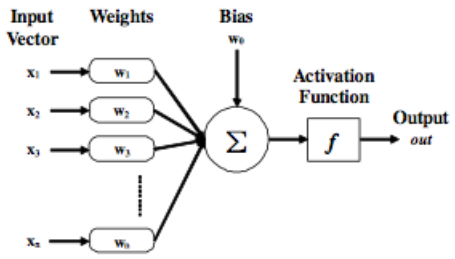


Figure 2.3: *The graphical representation of perceptron, this perceptron includes a total of n inputs. The output is calculated by sum up all the inputs and then passes through a activation function.*

Suppose the activation function $f(\cdot)$ is applied to the sum of these connected weights. Then a mathematical representation of the output y is given in (2.2.22)

$$y = f(\sum_{i=0}^{n} \omega_i x_i) \tag{2.2.22}$$

where x_i indicates the i^{th} input and ω_i demonstrates the weight for the i^{th} input.

After deciding its structure and initializing all the weights randomly, a perceptron can then be trained by giving a set of inputs and their corresponding outputs. Training is the process of adjusting all the weights along the direction of decreasing a cost function, which is defined to assess the error between estimated outputs and real outputs. In this way, it guarantees that the estimated outputs converge to the desire outputs gradually. This is the basic idea of supervised-learning algorithm. In general, perceptrons are used in binary classification problems. A sigmoid function is a good choice for the activation function because it guarantees the output values range from 0 to 1 and that helps to make the decision straightforward. Then Equation 2.2.22 can be illustrated as:

$$y = f(s) = \frac{1}{1 + e^{-s}} \tag{2.2.23}$$

where $0 < y < 1$, and $s = \sum_{i=0}^{n} \omega_i x_i$.

Many artificial intelligent tasks can be solved by extracting the right set of features and provide them to a machine learning algorithm. However, for many other tasks, they are difficult to know what features should be extracted. For example, suppose we want to detect bicycles in photographs. since we know that bicycles have wheels, then the presence of a wheel could be a feature. However, when shadows fall on a wheel, its image could be complicated and make it difficult to describe the wheel in terms of pixel values.

One approach to solve this problem is known as representation learning, which is proposed to discover both the mapping from representation to output and the mapping from representation to representation (Bengio *et al.*, 2015). This method gives AI systems more powerful learning capabilities in extracting good sets of features with minimal human intervention.

A typical example of a representation learning algorithm is auto-encoder. An auto-encoder consists of an encoder function and a decoder function; the encoder function has capability of converting the input data to another representation (in general, its a lower dimensional data). While the decoder function is used to convert the new data to the original representation. The purpose to do so is because autoencoder tries to hold as much as possible information and make the new representation has nice properties when an input run through encoder and decoder processes, such that the new representation can replace the original data to fulfill tasks.

Autoencoder was proposed by Rumelhart *et al.* (1985). An example framework of it is represented in Figure 2.4 (Ng, 2011). It is a three-layer net; the first layer L_1 includes a total of six nodes $\{x_1, x_2, ..., x_6\}$ to accept inputs, the hidden layer L_2 consists of three neurons in order to compress the input data to a new representation with lower dimensions, and the output layer L_3 has a same number of units with the input layer to store the output of the decoder function. The objective of training an auto-encode is to minimize the difference between the estimate output (decoder result) and the real output (it is the same as input data) through tuning weights between each two layers.

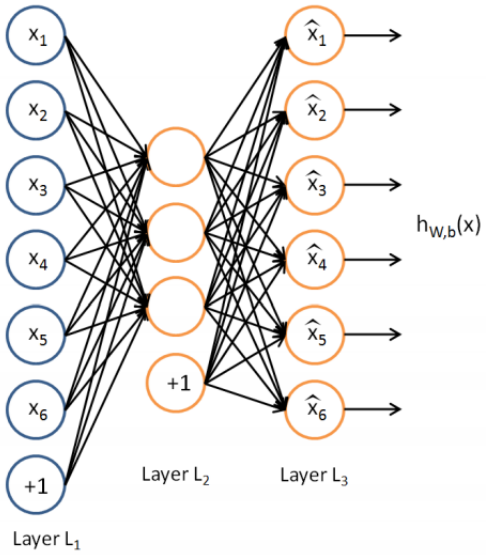


Figure 2.4: *A graphical representation of an auto-encoder neural network with three layers of elements. The training goal is to let the output approximate the input.*

In many real-world AI applications, a major difficulty is that variation of many factors affect every single piece of data that we can observe. Such as the shapes of a person are different from different viewing angle. Therefore, most applications require us to abandon those factors of variation that we do not care about. However, extracting such abstract features could be very difficult for raw data. One way to solve this problem is deep learning, which is a representation learning that expresses representation in terms of other simpler ones. Bengio *et al.* (2015) gives an example on explaining how a deep learning system combine simpler concepts in representing the concept of an image of a person. A corresponding image interpretation is given in Figure 2.5.

The function is applied to identity an object from a group of pixels, it is not easy for a computer to comprehend the raw input data. Deep learning overcomes this challenge by breaking the complicated mapping into a set of simple mappings and describes each of them by different layers of the model. For example, in Figure 2.5, the visible layer,the input of the model, indicates what we can

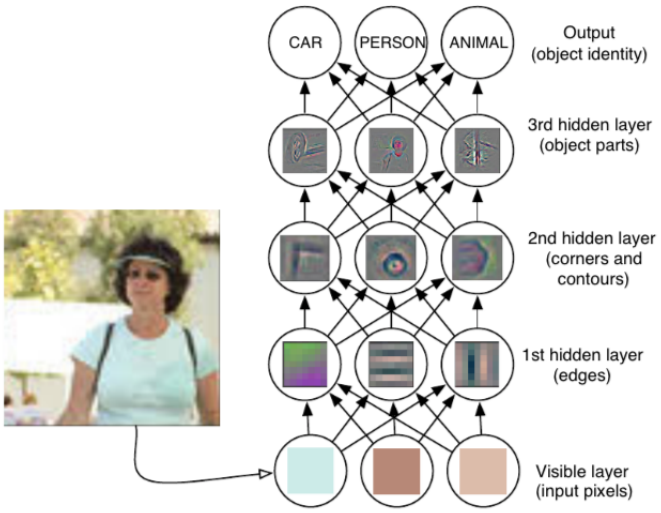


Figure 2.5: *A graphical representation of deep learning model.*

observe; three hidden layers are used to extract information about edges, corners and contours, and object parts, respectively. The extracting rule is that the next hidden layer information is extracted by collecting several parts of previous layer features such as the corner and contours are recognizable as collections of edges. Through this method, the image description can be recognized at the output layer.

2.2.2. MULTILAYER PERCEPTRON

Multi-layer perceptron, a mathematical function mapping a set of input values to output values, is a typical example of deep learning model. Multi-layer perceptron is an extension of a simple perceptron and it maps a sets of input data onto a set of appropriate outputs with multiple hidden layers. A general structure of multi-layer perceptron is shown in Figure 2.6 (Nakanishi). It illustrates that this multi-layer perceptron consists of three layers: input layer, hidden layer, and output layer. All the single nodes located in the previous layer are connected with all its next neighbour layer nodes through directed lines. In general, the previous layer's data vector is transferred to its subsequent layer through an activation function layer-by-layer until it reaches its output layer. Hence, multi-layer perceptrons are also called feed-forward networks.

An activation function defines the output of a node by giving an input or a set of inputs. It is usually an abstraction representing the action potential firing rate in a cell. In practice, sigmoid non-linear activation functions are widely used in modeling non-linear processes. Two types of sigmoid functions, hyperbolic tangent and logistic sigmoid functions, are given in (2.2.24) and (2.2.25). It is clear that the logistic function generates outputs range from 0 to 1, and the hyperbolic tangent

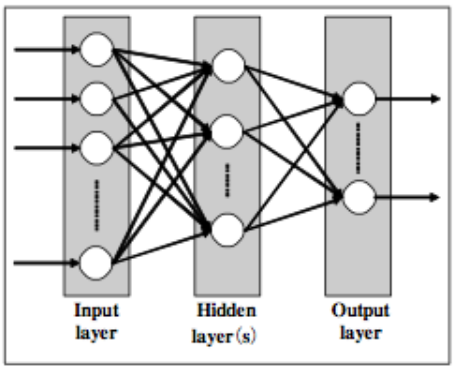


Figure 2.6: *Graphical representation of a general MLP structure*

function gives outputs range from -1 to 1. Different activation functions will be used according to different applications.

$$\sigma(x) = \tanh(x) \tag{2.2.24}$$

$$\sigma(x) = (1 + e^{-x})^{-1} \tag{2.2.25}$$

One of the major contribution of multi-layer perceptron is that it leads to the Back-Propagation (BP) algorithm, which was proposed by Werbos (1994) and Rumelhart *et al.* (1988), and which is also the main approach for training many neural networks.

Back-propagation. The back-propagation algorithm is used to compute gradients in neural networks in terms of the target function (usually indicate as the loss function) differential. When training a neural network, the cost function derivative with respect to each of the weights is computed to determine the effect of that weight. For example, the partial derivation of training criterion C with respect to θ indicates that whether the value of θ should be increased or decreased in purpose of decreasing C. In general, back-propagation is thought as the whole training method of a neural network rather than the network itself. Actually it is only the procedure of computing the gradients in that network. In order to apply the back-propagation method to generate the minimum costs, two other procedures must be applied, forward propagation to estimate output and cost computation as a training criterion have to be involved.

The detail procedure of training a neural network will be introduced by the example shown in Figure 2.6. Suppose $x_i^{l_1}$ indicates the i^{th} input of layer l_1, $x_i^{l_2}$ indicates the i^{th} input of layer l_2, $y_j^{l_3}$ demonstrates the j^{th} output of layer l_3, and $\omega_{ij}^{l_{12}}$ refers to the weight between i^{th} node in layer l_1 and j^{th} node in layer l_2. b^{l_1} and b^{l_2} indicate bias in layers l_1 and l_2, respectively.

- Forward Propagation

Given a set of training data x_i^1, the input vector for each hidden layer can be decided by formula (2.2.26). It claims that the j^{th} input of layer $l+1$ is equal to the sum of all the elements contained in layer l multiplied by their corresponding weights.

$$x_j^{l+1} = \sum_{i=0}^{n} x_i^l \omega_{ij}^l + b^l \tag{2.2.26}$$

These calculated inputs then through an activation function generate the outputs of all the nodes. This process is mathematically described as formula (2.2.27). Repeating the process layer-by-layer until the outputs $h_{W,b}(x)$ of the final layer are obtained.

$$y_j^{l+1} = f(x_j^{l+1}) \tag{2.2.27}$$

$$h_{W,b}(x) = y_j^{l+1} \tag{2.2.28}$$

- Cost Computation

  Assume we have a number of m training examples$(x^1,\ y^1),...(x^m,\ y^m)$. For each single example $(x,\ y)$, the squared-error cost function is defined as (UFL):

$$J(W,\ b;\ x,\ y) = \frac{1}{2} \parallel h_{W,b}(x) - y \parallel^2 \tag{2.2.29}$$

  Then the overall cost function for above training set is:

$$J(W,b) = [\frac{1}{m} \sum_{i=1}^{m} J(W,b;x^{(i)},y^{(i)})] + \frac{\lambda}{2} \sum_{l=1}^{n_l-1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} (W_{ji}^{(l)})^2 \tag{2.2.30}$$

  The first term in Equation 2.2.30 is the average sum-of-squares error. The second term is used to decrease the weight's magnitude, so it is called a regularization term or a weight decay term. In addition, it also helps prevent over-fitting. The term λ has capability of regulating the relative importance of these two terms.

- Back Propagation

  The goal of training a neural network is to minimize the cost function $J(W,b)$ with respect to parameters W and b by using an optimization algorithm such as gradient decent. Since $J(W,b)$ is a non-convex function, so it may be affected by local optima; but it usually works well in practice.

  First we need to calculate the partial derivatives of the overall cost function with respect to weights and bias.

$$\frac{\partial}{\partial W_{ij}^{(l)}} J(W,b) = [\frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial W_{ij}^{(l)}} J(W,b;x^{(i)},y^{(i)}] + \lambda W_{ij}^{(l)} \tag{2.2.31}$$

$$\frac{\partial}{\partial b_i^{(l)}} J(W,b) = \frac{1}{m}\sum_{i=1}^{m} \frac{\partial}{\partial b_i^{(l)}} J(W,b;x^{(i)},y^{(i)}] \tag{2.2.32}$$

Then the parameters W and b can be updated by one iteration of gradient descent as:

$$W_{ij}^{(l)} = W_{ij}^{(l)} - \alpha \frac{\partial}{\partial W_{ij}^{(l)}} J(W,b) \tag{2.2.33}$$

$$b_i^{(l)} = b_i^{(l)} - \alpha \frac{\partial}{\partial b_i^{(l)}} J(W,b) \tag{2.2.34}$$

The overall training starts by initializing the parameters randomly. According to these parameters, it performs feed forward pass for a training set such that it computes the activations layer-by-layer up to the output layer. Then it calculate a cost function, which indicates the sum of square errors for all the training examples. Finally, in order to decrease the difference between the estimated output and the real output, the gradient descent optimization algorithm is applied to minimize the cost function. Repeat this procedure until the value of the cost function is within a specific area, then the neural network is well-trained.

As multi-layer perceptron, auto-encoder is trained by using the back-propagation algorithm as well. The difference is that the output is not given directly, but the same with the input data. The training data is transferred layer-by-layer to the output layer. Then the difference between the estimated output and the input is compared, and the control goal is to minimize their difference by tuning weights between layers.

2.2.3. STACKED AUTO-ENCODER

A more complicated network structure, the stacked auto-encoder (or deep auto-encoder), is introduced in this section. It has stronger representation power in dealing with large set of non-linear functions than shallower networks. It was explained in Subsection 2.2.1 in an image recognition application example; which indicates that the first hidden layer only has capability of recognising the outline of the image, and the detail information is detected by those following hidden layers.

Difficulty arises when training a stacked auto-encoder since the traditional BP algorithm would not work efficiently anymore. Three reasons are represented in UFL to explain it.

- **Data Providing Limitations**

  Back propagation is a learning algorithm that demands for a quantity of labeled data. However, labeled data can not be provided all the time. Moreover, it is difficult to represent a complex object by using labeled data, because labeled data has limited capabilities such that

it can not describe all the characteristics of a kind of object. Thus, labeled data gives a low level expressive capability and has high probability of causing over fitting.

- **Local Optima Limitation**

  Back propagation is a gradient descent algorithm that aims to minimize a cost function according to tuning weight parameters. However, deep network involves in a highly non-convex optimization problem that make it easy to run into local minimum and missing the global optima for the randomly initialization of weights.

- **Gradients Diffusion Limitation**

  In backward propagation procedure, the gradient is rapidly decreased from output layer to input layer along all the hidden layers. It leads to the weights in those previous hidden layers change slightly at each iteration bring about that they are short of learning capabilities. Then the net weights have to be updated again and again. This is so called "gradients diffusion".

Greedy Layer-wise Training For neural networks with many hidden layers and a number of adaptive connections, most of the learning algorithms for training shallow networks are not capable to discover appropriate weights for each layer. An efficient way of training deep network is called greedy layer-wise training, whose proposal is training the weight parameters layer-by-layer. The basic idea is taking each layer together with its subsequent layer form a simple auto-encoder, and train each of them separately using back-propagation algorithm. For example, taking the input and first hidden layers form the first auto-encoder at beginning, whose real output is equal to the input. Apply the back-propagation algorithm and train it and then collect the estimated output of the first hidden layer, which in turn is the input and output of the second auto-encoder. Iterate this process until all the layers have been transversed.

An example of a stacked auto-encoder is shown in Figure 2.7 and is used to explain the greedy layer-wise algorithm. To make it simple, the neural network is designed with only one hidden layer.

This neural network starts to train by forming the first auto-encoder with the input layer and the hidden layer. The formed auto-encoder is given in Figure 2.8. The estimated output $[\hat{x}_1; \hat{x}_2; \hat{x}_3; \hat{x}_4]$ compares with the input training data such that obtain a cost function; Then the back propagation algorithm is applied to train the auto-encoder. After well-training the auto-encoder, the third layer will be deleted, and the output of the hidden layer is obtained by performing forward propagation. The obtained data set acts as the input of the second

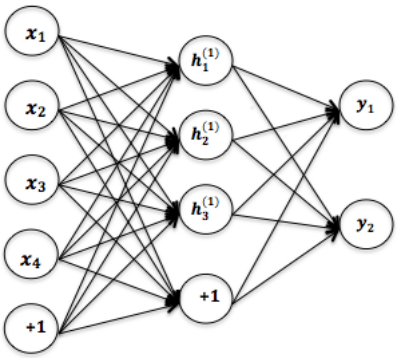


Figure 2.7: *An example stacked auto-encoder used to explain greedy layer-wise training algorithm*

auto-encoder.

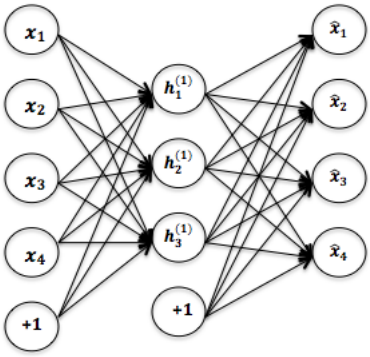


Figure 2.8: *The first auto-encoder formed by the input and the first hidden layer*

The second auto-encoder is formed by the hidden layer and the output layer. This auto-encoder is trained with the same method of the first auto-encoder such that it obtains the well-trained weights between these two layers.

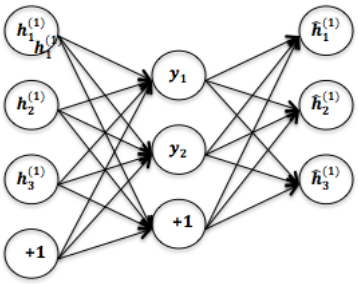


Figure 2.9: *The second auto-encoder formed by the first hidden layer and the output layer*

Finally, fine-tune procedure is always applied to get a better weights combination. This is implemented by forming a stacked auto-encoder with all layers included in the neural network, which is represented in Figure 2.10. It needs to recommend the weights between the third

layer and the fourth layer, as well as the final two layers are equal to the transposed weights of the pre-trained weights in their relative positions, which means $W3 = W2'$ and $W4 = W1'$. Then applying the back propagation algorithm the entire neural network is fine tuned. After tuning the weights, the final two layers are deleted and the original neural network in Figure 2.7 is used in the dimension-reduction problem.

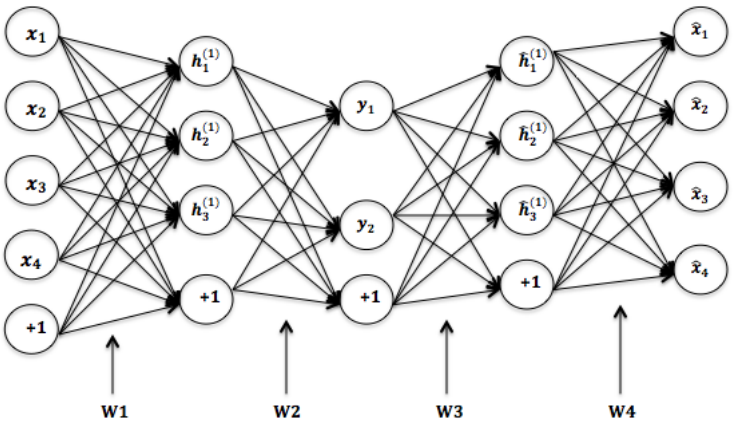


Figure 2.10: *Fine-tuning the pre-trained weights with stacked auto-encoder*

In this example, the stacked auto-encoder is used in dimension-reduction, we pre-train all the weights using the same method and fine-tune them by forming a stacked auto-encoder with all the layers. For some other applications, the weights between the final two layers can be trained using different methods such as those that receive a different output format. For example, the raw digit classification example introduced in UFL, the last weight combinations are trained by linear activations, and the fine-tune process is implemented by giving a desired output corresponding to the training data.

3

REINFORCEMENT LEARNING IN CONTROL

In this chapter, reinforcement learning experiment is implemented by involving in a segway setup in Section 3.1. A segway-balancing task is applied to check the performance of the standard actor-critic algorithm. Then, the high-dimension challenge is presented in Section 3.2. In order to overcome this challenge, the structure of deep reinforcement learning algorithm is proposed in Section 3.3.

3.1. SEGWAY-CONTROLLING EXPERIMENT

The segway setup will be used in this thesis is shown in Figure 3.1, consists of two parts: a wheel and a link.

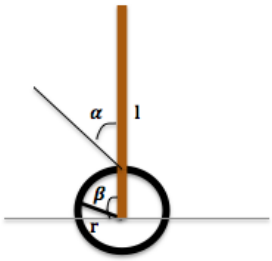


Figure 3.1: *imitated segway system*

The vector of the configuration variables is indicated as $q = [\alpha \ \beta]^T$, where α is the angle between the link and the vertical axis and β represents the wheel angle. Najafi *et al.* (2015) provides the

mathematical model of the system. The Euler-Lagrange equation of a Segway is:

$$M(q)\ddot{q} + C(q,\dot{q})\dot{q} + G(q) = B(q)\tau + F_{ext} \tag{3.1.1}$$

where $M(q) \in R^{n\times n}$ is the inertia matrix, $C(q,\dot{q}) \in R^n$ is the Coriolis and centrifugal forces matrix, $G(q) \in R^n$ indicates the gravity matrix, $B(q) \in R^n$ denotes the input vector and $F_{ext} \in R^n$ represents the external forces. These matrices are given by:

$$\begin{aligned}
M(q) &= \begin{bmatrix} m_b l^2 + I_b & m_b r l cos(\alpha) + M_1 \\ m_b r l cos(\alpha) + M_1 & (m_W + m_b) r^2 + I_W + M_2 \end{bmatrix} \\
C(q,\dot{q}) &= \begin{bmatrix} 0 & C_1 \\ -m_b r l \dot{\alpha} sin(\alpha) + C_2 & C_3 \end{bmatrix}, \qquad B(q) = \begin{bmatrix} -2 \\ 2 \end{bmatrix} \\
F_{ext} &= \begin{bmatrix} 2b_j(\dot{\phi} - \dot{\alpha}) \\ -2b_g - 2b_j(\dot{\phi} - \dot{\alpha}) \end{bmatrix}, \qquad G(q) = \begin{bmatrix} -m_b l g sin(\alpha) \\ G_1 \end{bmatrix}
\end{aligned} \tag{3.1.2}$$

with

$$\begin{aligned}
&M_1 = c m_b r l cos(\alpha - \phi), \quad M_2 = c^2 m_b r^2 + 2 c m_b r^2 cos(\alpha) \\
&C_1 = c m_b r l \dot{\phi} sin(\alpha - \phi), \quad C_2 = -c m_b r l \dot{\alpha} sin(\alpha - \phi) \\
&C_3 = -c m_b r^2 \dot{\phi} sin\phi, \quad G_1 = -c m_b r g sin(\phi)
\end{aligned} \tag{3.1.3}$$

The symbols are explained in Table 3.1. In this thesis, only the standard Segway model is used, which means that the link is connected with the center of the wheel. This results in the parameter $c = 0$. Then, the matrices in Equation 3.1.1 become:

$$\begin{aligned}
M(q) &= \begin{bmatrix} m_b l^2 + I_b & m_b r l cos(\alpha) \\ m_b r l cos(\alpha) & (m_W + m_b) r^2 + I_W \end{bmatrix} \\
C(q,\dot{q}) &= \begin{bmatrix} 0 & 0 \\ -m_b r l \dot{\alpha} sin(\alpha) & 0 \end{bmatrix}, \qquad B(q) = \begin{bmatrix} -2 \\ 2 \end{bmatrix} \\
F_{ext} &= \begin{bmatrix} 2b_j(\dot{\phi} - \dot{\alpha}) \\ -2b_g - 2b_j(\dot{\phi} - \dot{\alpha}) \end{bmatrix}, \qquad G(q) = \begin{bmatrix} -m_b l g sin(\alpha) \\ 0 \end{bmatrix}
\end{aligned} \tag{3.1.4}$$

The objective of this task is to balance the link of the Segway to the upright position. The state x that used to describe the states of the system includes four elements: link angle (α), link angular

Model parameter	Symbol	Value	Units
Link inertia	I_b	$2.60 \cdot 10^{-2}$	Kgm^2
Link mass	m_b	2.55	Kg
Wheel inertia	I_W	$4.54 \cdot 10^{-4}$	Kgm^2
Wheel mass	m_W	$3 \cdot 10^{-1}$	Kg
Gravity	g	9.81	ms^{-2}
Link half length	l	$13.8 \cdot 10^{-2}$	m
Wheel radius	r	$5.5 \cdot 10^{-2}$	m
Rolling friction	b_g	$6.5 \cdot 10^{-3}$	Nms
Joint friction	b_j	$1 \cdot 10^{-4}$	Nms
Torque constant	K_t	$3.15 \cdot 10^{-2}$	NmA^{-1}
Back EMF	k_e	$3.15 \cdot 10^{-2}$	NmA^{-1}
Rotor resistance	R_m	5.21	Ω
Gear ratio	N_g	35	-
Input gain	N_s	25	V

Table 3.1: *Physical parameters of the segway system*

velocity ($\dot{\alpha}$), wheel angle (β) , and wheel velocity ($\dot{\beta}$), the state x is defined as

$$x = \begin{bmatrix} \alpha \\ \beta \\ \dot{\alpha} \\ \dot{\beta} \end{bmatrix} \tag{3.1.5}$$

The reward function r is defined by a continuous quadratic function:

$$r(x_{k+1}, u_k) = -x_{k+1}^T Q x_{k+1} - u_k^T P u_k \tag{3.1.6}$$

where Q and P are positive weighting matrices. They are used to weighted the angel and angular velocity of the link and the wheel.

As described in Equation 2.1.3, the value function and the policy are approximated in linear by using a group of radial basis functions.

$$V_\theta(x) = \theta^T \phi(x) \tag{3.1.7}$$

$$\pi_\vartheta(x) = \vartheta^T \phi(x) \tag{3.1.8}$$

with $\phi(x)$ is a column of normalised RBFs $\phi_i(x)$.

A challenge that arose when applying actor-critic methods was the trade-off between exploitation and exploration (Sutton and Barto, 1998). Exploitation means the actions that have been tried in the past and produce effective rewards are prefer to be chosen. While exploration suggests that some new actions have to be selected such that make a better action in the future. Exploitation has ability of maximizing the expected reward, while exploration may produce a better long-term total reward. Thus, the balancing between exploitation and exploration is an important factor to be taken into consideration. Sutton and Barto (1998) introduced several simple ways of balancing them, like the ε-greedy methods and the softmax methods. However, they cannot solve all the reinforcement learning problems. In this experiment, a simple way is used to realize exploration. For each trial, it explores once for each three time-steps. By using this method, it achieves the control goal gradually.

The experiment is divided into two stages; the first one is the link-only control (Subsection 3.1.1), which means only the states of the link are taken into consideration. The second stage is the link and wheel velocity control (Subsection 3.1.2), in which also the velocity of the wheel is cared about. Since the position of the wheel is not so important in real world and it requires much more time than the previous stage we will not consider this variable in this experiment.

3.1.1. LINK-ONLY CONTROL

In this experiment, only the angle and the angular velocity of the link is considered, then the state x is defined as

$$x = \begin{bmatrix} \alpha \\ \dot{\alpha} \end{bmatrix} \tag{3.1.9}$$

where $\alpha \in [-\pi, \pi]$ and $\dot{\alpha} \in [-25, 25]$. The quadratic reward function aims to generate a maximum value in the upright position $[0 \;\; 0]^T$, the weighted matrices of Q and P shown in Equation 3.1.6 are defined as:

$$Q = \begin{bmatrix} 3 & 0 \\ 0 & 0.3 \end{bmatrix} \qquad P = 0.01 \tag{3.1.10}$$

In this experiment, a total of 900 RBFs are used to paramiterize both the value function and the policy, separately. The width of RBFs should be defined such that the plot of these 900 RBFs smooth. According to this criterion, width matrix B in Equation 2.1.21 is defined as:

$$B = \begin{bmatrix} 0.035 & 0 \\ 0 & 2.5 \end{bmatrix} \tag{3.1.11}$$

The plot of RBFs with above width is shown in Figure 3.2. One thing that needs to be mentioned is that we just care about the shape of RBFs plot but not their values in this thesis.

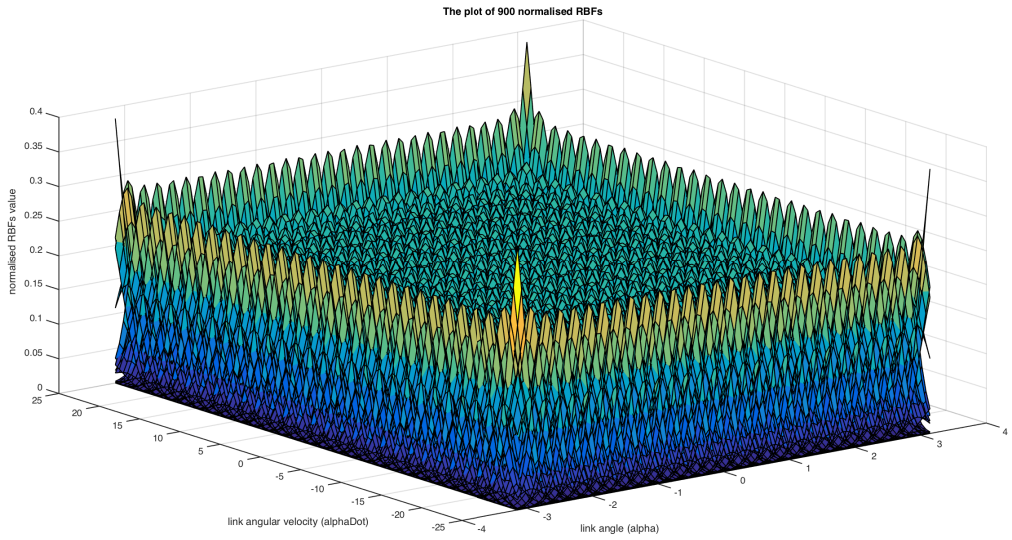


Figure 3.2: *Graphical representations of normalised RBFs plot for alpha and alphaDot*

Giving an initial state $[-\pi/8 \;\; 0 \;\; 0 \;\; 0]^T$, a random input $u \in [-5, \;\; 5]$, and list all the relative parameters in Table 3.2, The link of the Segway is balanced gradually.

RBFs_num	trial_length	sampling time	γ	variance	α_a	α_c
900	8s	0.02	0.97	5e-2	0.0035	0.5

Table 3.2: *List of parameters values*

with

- RBFs_num: the number of RBFs for each variable
- trial_length: time spending of each trial
- sampling time: time step
- γ: the discount term relates to reward function
- variance: the variance of the random signal.
- α_a & α_c: learning rates for actor and critic, respectively.

The critic and actor mappings are shown in Figure 3.3. From the critic figure, it is clear that the highest value locates around the point $[0,0]^T$, and it decreases when away from that point. From the actor figure, it shows that actions are limited within area $[-5,5]$, and their absolute values increase when the link angle far away from 0. This is because the system requires more power to balance the link when it is far away from the upright position and less power needed when it closes to the objective position.

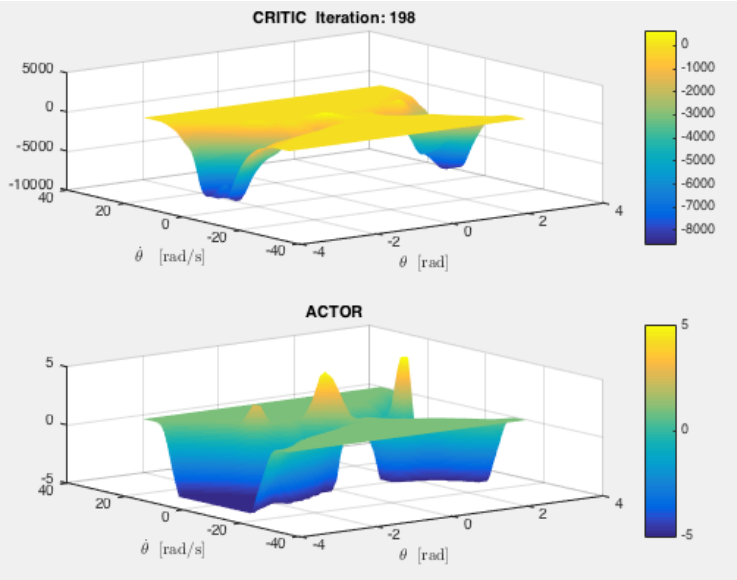


Figure 3.3: *Graphical representations of critic and actor mappings*

A total of 200 trials are used to learn to control this system. For each trial, it needs 0.703s. Then it requires about 150s to implement the entire learning process, Figure 3.4 represents the sums of rewards and the temporal difference for each trial.

According to Figure 3.4, we can see that after 80 trials, the system obtains its maximum rewards sum and the temporal difference converge around zero. However, it is not equal to zero. This is because the axle friction generates a torque and in order to neutralize the torque, the gravity needs to generate an opposite torque. Meanwhile, the wheel has to conquer this torque and the torque generated by the rolling friction once it is moving. Figure 3.5 represents the angle and angular velocity of the link. It indicates that the final angle and angular velocity of the link are close to zero but not equal to zero either. In order to keep the link balanced, the wheel has to keep moving all the time. This is confirmed in Figure 3.6.

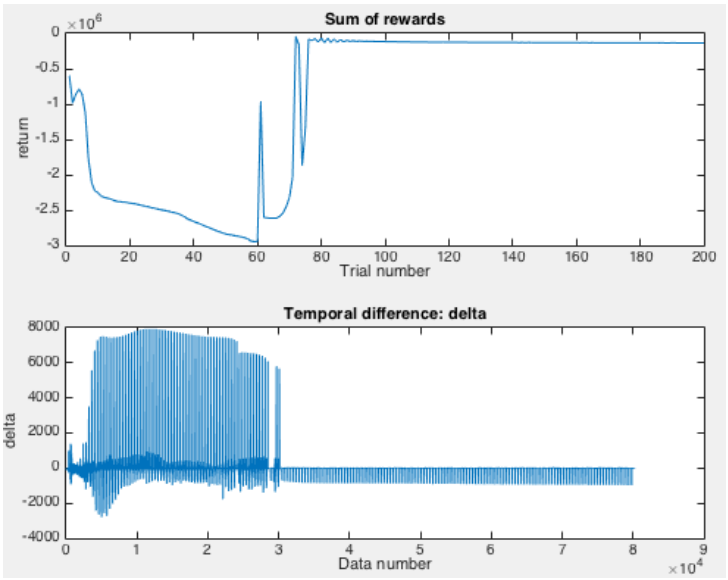


Figure 3.4: *Results of rewards sums and temporal differences*

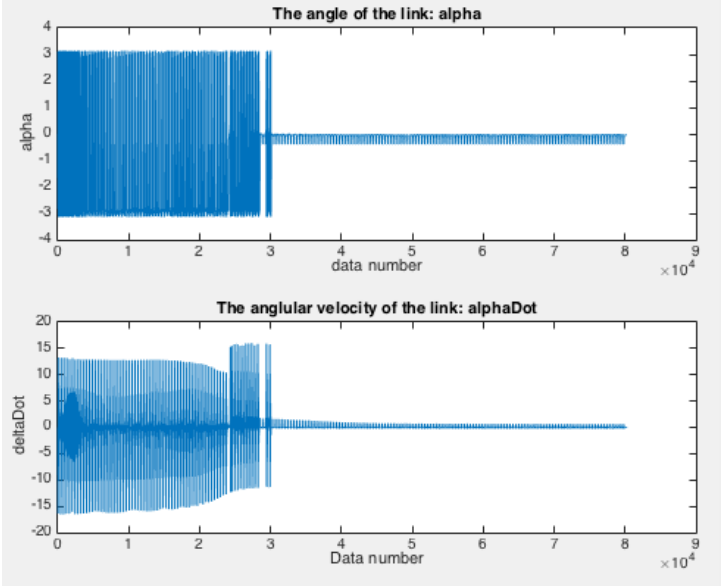


Figure 3.5: *Angle and angular velocity of the link*

3.1.2. LINK AND THE WHEEL VELOCITY CONTROL

In this section, the wheel angular velocity $\dot{\beta}$ is also taken into consideration. Then the state that used to update the reward function, the value function and the policy becomes

$$x = \begin{bmatrix} \alpha \\ \dot{\alpha} \\ \dot{\beta} \end{bmatrix} \tag{3.1.12}$$

where $\alpha \in [-\pi, \pi]$, $\dot{\alpha} \in [-25, 25]$,and $\dot{\beta} \in [-25, 25]$.

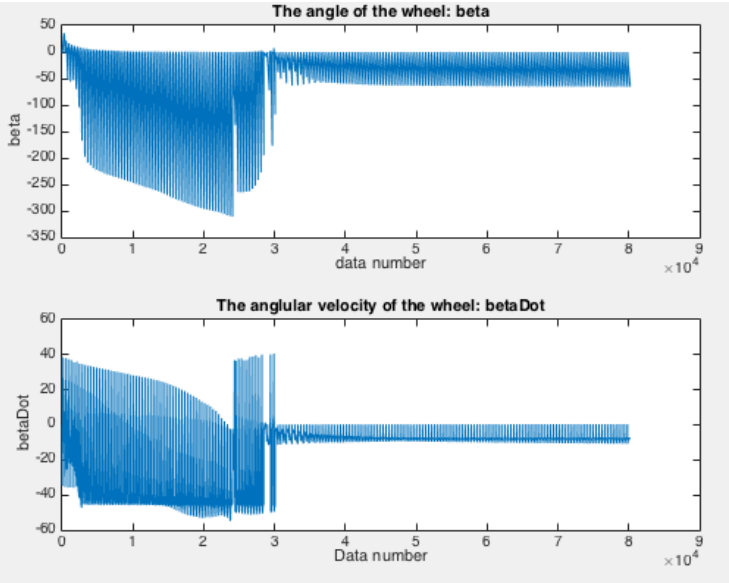


Figure 3.6: *Angle and angular velocity of the wheel*

In this experiment, a number of 27000 RBFs are used to parameterize the value function and the policy, respectively. Since the previous two variables are still link angle and angular velocity, so the widths of RBFs relative to these two variables are the same with Equation 3.1.11. In addition, since the range of wheel angular velocity is the same as the link angular velocity, so the RBF width for this variable can be 2.5 as well. The RBFs plot of the link angular velocity and the wheel velocity is shown in Figure 3.7, which indicates that the plot of RBFs is smooth.

$$B = \begin{bmatrix} 0.035 & 0 & 0 \\ 0 & 2.5 & 0 \\ 0 & 0 & 2.5 \end{bmatrix} \tag{3.1.13}$$

The weighted matrices in Equation 3.1.6 are shown in Equation 5.2.1.

$$Q = \begin{bmatrix} 3 & 0 & 0 \\ 0 & 0.1 & 0 \\ 0 & 0 & 0.1 \end{bmatrix} \qquad P = 0.01 \tag{3.1.14}$$

Giving an initial state $[-\pi/8 \quad 0 \quad 0 \quad 0]^T$, a random action $u \in [-5, \quad 5]$, and list all the relative parameter values in Table 3.3, the link of the Segway is balanced finally.

RBFs_num	trial_length	sampling time	γ	variance	α_a	α_c
27000	8s	0.02	0.97	5e-2	0.0035	0.5

Table 3.3: *List of parameters values*

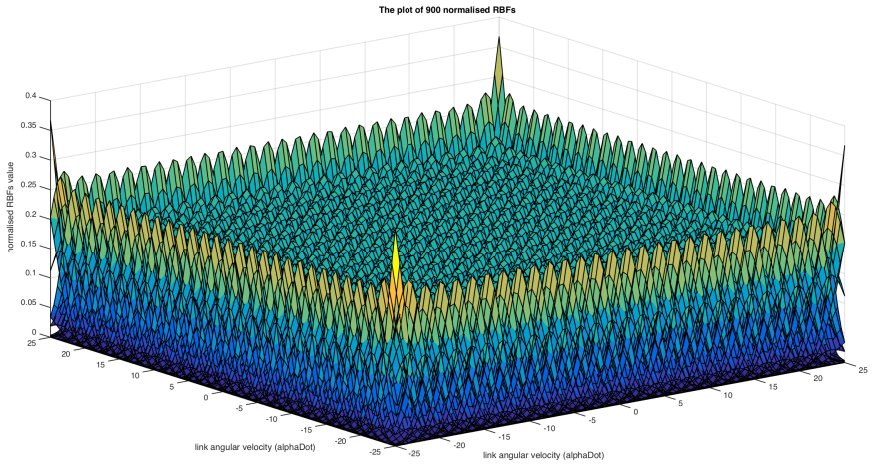


Figure 3.7: *Graphical representations of normalised RBFs plot for alphaDot and betaDot*

From Figure 3.8, we can see that a total of 800 trials are set to try to control the Seyway (for each trial, it needs 4.452s, then it costs about one hour to implement the entire learning process). It also indicates that the system obtains its maximum rewards sum and the temporal difference converge around zero after 400 trials. Since the same reason with above system, the angle and angular velocity of the link do not equal to zero either, this is indicated in Figure 3.9. At the same time, the wheel has to keep moving to keep the link balanced, as shown in Figure 3.10.

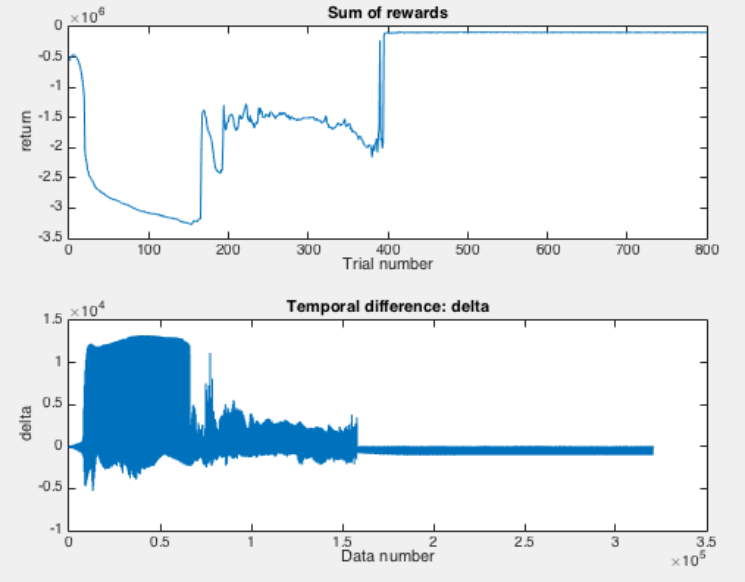


Figure 3.8: *Results of reward sums and temporal different value δ*

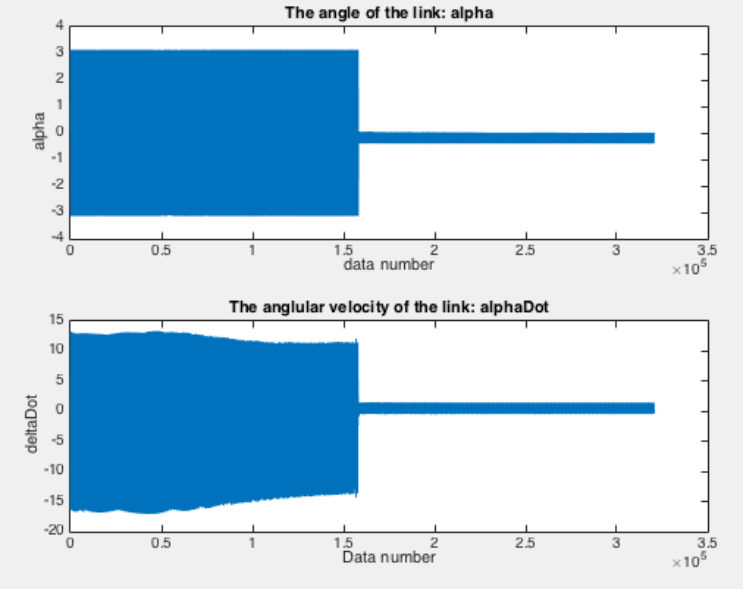


Figure 3.9: *Final results for angle and angular velocity of the link*

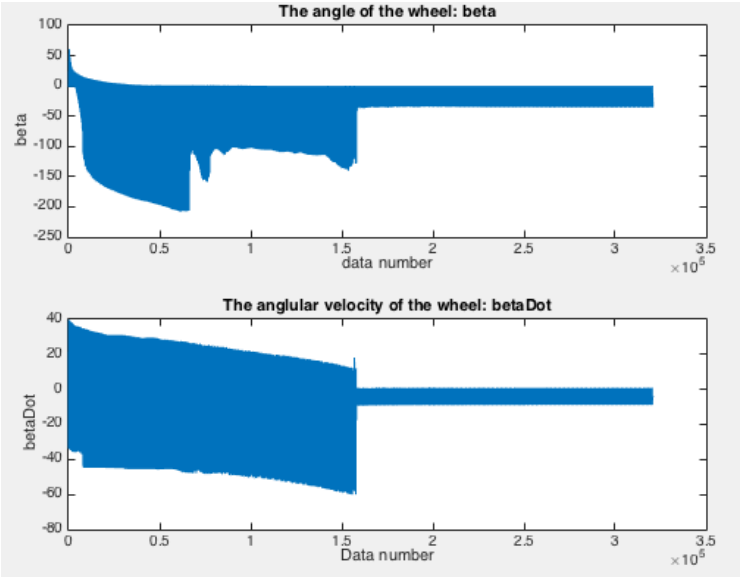


Figure 3.10: *Final results angular velocity of the link*

3.2. HIGH-DIMENSION CHALLENGE IN RL

Even through the wheel velocity is not considered in this project, we did some tests on it also. In this case, the state is defined as:

$$x = \begin{bmatrix} \alpha \\ \dot{\alpha} \\ \beta \\ \dot{\beta} \end{bmatrix} \tag{3.2.15}$$

A total of 810000 RBFs are used to parameterize each variable. In this situation, it needs 156.46s to run each trial, which is much longer than the previous two experiments. Moreover, in order to fulfill the learning process, many more trials are needed as well. Therefore, It becomes quite difficult and inefficient to control this system.

As most of robots are designed to implement complex tasks and inherently continuous, so the actions and states are continuous and with high dimensions. Then the robot system has to be designed to deal with such high-dimensional states and actions. Kober *et al.* (2013) analysed the state and action spaces of a ball paddling robot, as shown in: Figure 3.11.

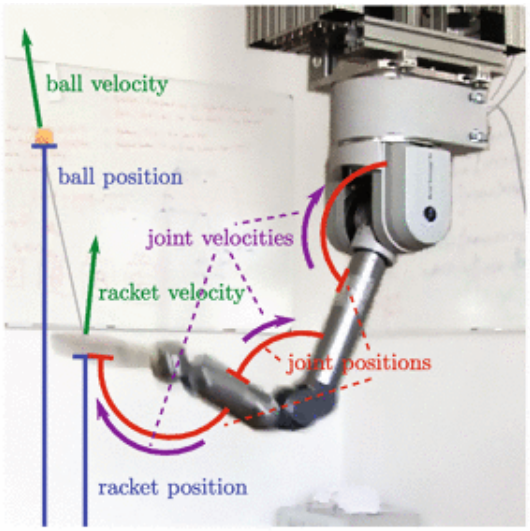


Figure 3.11: *A graphical illustration of the a robot reinforcement learning task state space.*

Figure 3.11 indicates that the robot's states consist of its joint angles and velocity for each of its seven degrees of freedom as well as Cartesian position and velocity of the ball. The actions of the robot would be the generated motor commands which often are torques or accelerations. Then for the ball paddling robot, the state-dimension is $2 \times (7+3) = 20$ and the continuous action-dimension is 7. Note that the 10-30 dimensional continuous actions common in robot reinforcement learning are considered large (Powell, 2012). This ball paddling robot is designed to implement just one specific task, which is hiting the ball to a specific area. However, higher dimensional state and action spaces are required in cases of implementing more complicated or multiple tasks. Especially, human-like actuation which allows the control of stiffness. (Nakanishi *et al.*, 2008) proposed a method to reduce the high dimensional state space of the ball paddling robot. The main idea is that controlling the robot in racket space with an operational space control law instead of the original space. However, Schaal *et al.* (2002) and Nakanishi *et al.* (2008) found, by experience, that this method limits the dynamic capability of the robot.

3.3. HIGH-DIMENSION CHALLENGE SOLVE METHOD

In this project, we try to use deep learning algorithm to solve this problem. Since the difficulty is cursed by high-dimensional state spaces, then methods that have capability of reducing dimensions could be helpful in overcoming this challenge. Deep neural networks have been well-used in representation learning in the computer science field, especially in pattern recognition, e.t. Here, we would like to apply them in reducing the dimension of state spaces in control area. The basic idea is applying a deep neural network into reinforcement learning to reduce the dimension of system states, and then according to the low-dimensional representation control the original system. Taking the ball paddling robot as an example, the general method of integrating deep neural network into reinforcement learning is described in Figure 3.12.

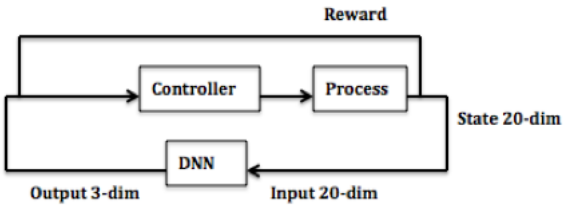


Figure 3.12: *A graphical illustration of integrating a DNN into RL*

From Figure 3.12, we can see that the new group of states generated by the system are transferred to a deep neural network architecture first, and then the outputs of DNN, which is a low-dimensional representation, are given to the controller to generate new actions for the process. In this ball paddling system, we suppose a DNN that can reduce the original 20-dimensional state space into a three dimensional representation. It is apparent that if this method works the learning process becomes faster and more efficient than before.

4

SAE IN DIMENSION REDUCTION

In this chapter, the stacked auto-encoder described in Subsection 2.2.3 is used in this chapter. In order to improve efficiency, the SAE is trained offline. Once an auto-encoder is well-trained, it will be integrated into reinforcement learning algorithm to reduce the state dimensions online. Section 4.1 introduces the selections of training data and experiment parameters, as well as provides the results of the stacked auto-encoder. Section 4.2 validates the well-trained model by using a different reference signal.

4.1. STACKED AUTO-ENCODER EXPERIMENT

4.1.1. STACKED AUTO-ENCODER ALGORITHM

According to Subsection 2.2.3, the stacked auto-encoder algorithm used in this experiment can be described in 2.

4.1.2. TRAINING DATA COLLECTION

In order to obtain the state of the Segway with four dimensions, an observer feedback controller is designed to control it, Simulink modes of the segway system and the feedback controller are given in Figure 4.2 and Figure 4.3, respectively. The reference is shown in Figure 4.1. With a sampling time of 0.001, a training data set with a size of 20242×4 is collected after 20s.

Algorithm 2 SAE algorithm

Require: : Layer number l, pre-train iteration number I, fine tune iteration number F training data set of l^{th} auto-encoder x_i^l, weight between l and l+1 layers W_{ij}^l, learning rate α.

% pretrain model using stacked auto-encoders

Start from $l \leftarrow 1$

loop

  Randomly choose W_{ij}^l

  Start from $I \leftarrow 1$

  loop

    Implement forward propagation: $y_j^l = g(\sum_{i=0}^{n} x_i^l w_{ij}^l + b^l)$

    Cost computation: $J(W,b) = [\frac{1}{m}\sum_{i=1}^{m} J(W,b;x^{(i)},y^{(i)})] + \frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_{l+1}}(W_{ji}^{(l)})^2$

    Apply back-propagation: $\frac{\partial}{\partial b_i^{(l)}} J(W,b) = \frac{1}{m}\sum_{i=1}^{m}\frac{\partial}{\partial b_i^{(l)}} J(W,b;x^{(i)},y^{(i)}]$,
    $W_{ij}^{(l)} = W_{ij}^{(l)} - \alpha\frac{\partial}{\partial W_{ij}^{(l)}} J(W,b), \quad b_i^{(l)} = b_i^{(l)} - \alpha\frac{\partial}{\partial b_i^{(l)}} J(W,b)$

  end loop

end loop

% Fine tune model using gradient descent

Start from $F \leftarrow 1$

loop

  [reconX,mappedX]= Run data through auto-encoder: run forward propagation layer-by-layer until final layer. (reconX is the final layer output, mappedX is the middle layer output).

  Calculate cost function and then implement back-propagation.

end loop

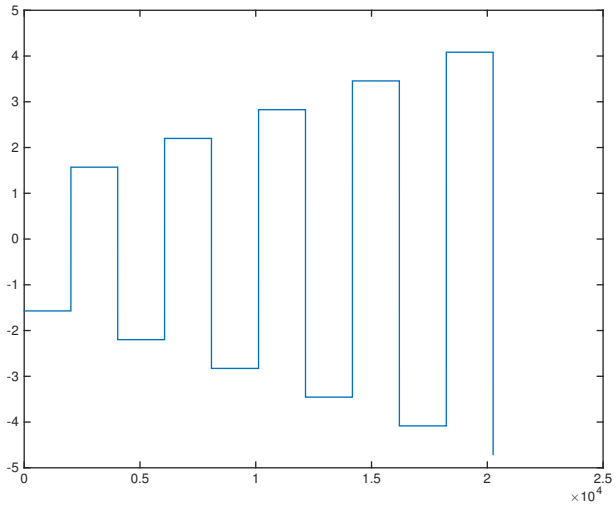


Figure 4.1: *Reference of the system*

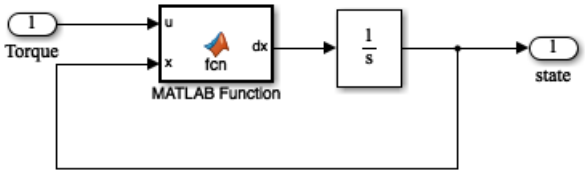


Figure 4.2: *Simulink model of the Segway model*

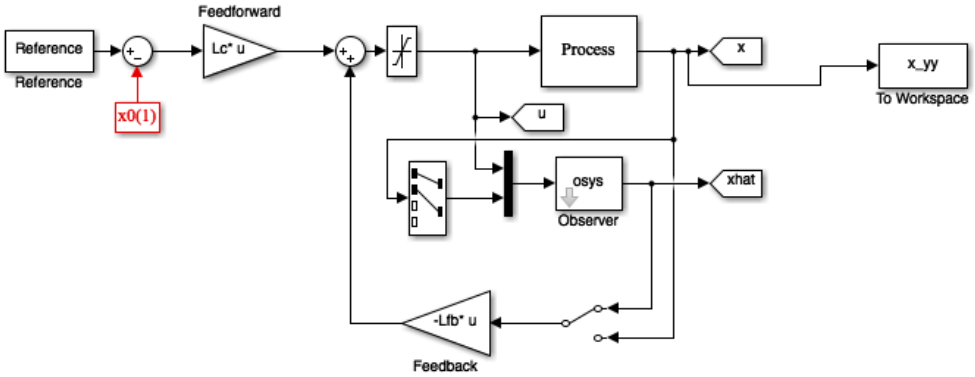


Figure 4.3: *Simulink model of the feedback controller*

4.1.3. SELECTION OF PARAMETERS

The toolbox written by Van der Maaten *et al.* (2007) is used in this thesis to reduce the four-dimensional state space of the segway system. Some important parameters are decided in this section.

Learning rate Learning rate is a parameter deciding the changing rate of neural network weights. It is indicated as parameter α in Equation 2.2.33. If α is too small, it results in slow conver-

gence; If α is too large, the cost function may not decrease on every iteration, then it may not converge. So the selecting of the learning rate is very important in training a neural network. To keep the cost decreasing properly, a learning rate of 0.001 is choosing in this project.

Iteration number Iteration number is another factor that affects the results of neural networks. If the number of iterations is too small, it cannot obtain the optimal cost; If the iteration number is too large, it may cause over-fitting problem, which means the well-trained model only generates good results for the training data, but not for other data. In condition of avoiding these problems, 120 and 156 iterations are chosen for pre-training and 200 iterations are selected for fine-tune process after many times testing.

Number of layers In order to choose a proper structure for the stacked auto-encoder, a table with different layer numbers and their corresponding performances are represented below:

Table 4.1: The relationship between the layer numbers and their performances

Structure of SAE	Pre-training iterations	fine-tune iterations	Final error
4-3-4	130	200	0.67629
4-5000-3-5000-4	130/130	200	0.48066
4-50-100-3-100-50-4	120/200/150	200	56.5727
4-100-200-400-3-400-200-100-4	96/130/150/120	200	59.4195

From this table, we can see that the stacked auto-encoder with three hidden layers performs better than other structures. In addition, Table 4.2 indicates that the performance is improved by increasing hidden layer units of the $4-50-100-3-100-50-4$ structure. However, their performances are still worse than the three hidden layers structure. Therefore, the structure with three hidden layers is used in the experiment.

Table 4.2: The relationship between the node numbers and their performances

Structure of SAE	Pre-training iterations	fine-tune iterations	Final error
4-50-100-3-100-50-4	120/200/150	200	56.5727
4-500-5000-3-5000-500-4	115/90/130	2000	1.1663
4-3000-300-3-300-3000-4	103/109/160	2000	1.0291

Number of hidden-layer nodes The last important parameter that needs to be decided is the number of nodes in the first hidden layer. In the same way, a table indicating the relationship

between the number of the first hidden layer nodes and their performance is given below.

Table 4.3: The relationship between the number of nodes and their performances

number of nodes	Pre-training iterations	fine-tune iterations	Final error
800	130/200	200	1.0121
1500	96/130	200	0.77594
5000	115/130	200	0.48066
6000	115/150	230	0.47232
7000	115/150	200	0.44491
10000	120/156	151	0.36524

Table 4.3 shows that with the increasing number of the first hidden layer units, the performance of the stacked auto-encoder improved. Taking the computer capability and efficiency into consideration, the number 10000 is chosen eventually. Thus, the final structure of the stacked auto-encoder is $4-10000-3-10000-4$. In addition, their MSEs for all these structures are given in Figure 4.4. This figure indicates that errors decrease rapidly first, and then slowly until reach their minimum values, this is corresponding to the gradient decent theory.

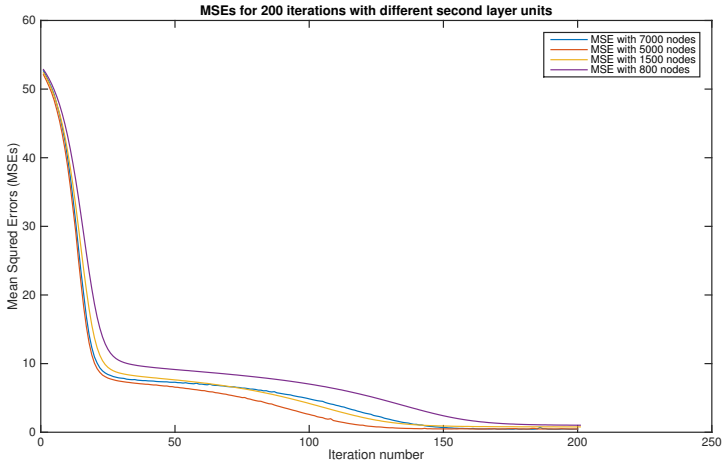


Figure 4.4: *MSEs for fine tune process*

To make it clear, all the parameters are listed in Table 4.4,

Table 4.4: Parameter values used in the SAE

learning rate α	AE1 Pre-training iter	AE2 Pre-training iter	fine-tune iter	SAE structure
0.0001	120	156	151	$4-10000-3-10000-4$

4.1.4. TRAINING RESULTS

Using 2 and parameters described above, the stacked auto-encoder is well-trained with time shown in Table 4.5 with AE1_iter_time, AE2_iter_time, and fine_tune_iter indicate the time spending for each iteration in the first auto-encoder, the second auto-encoder, and the fine_tune process, respectively.

Table 4.5: Time-spending of the SAE

AE1_iter_time	AE2_iter_time	fine_tune_iter_time	final error
10s	650s	720s	0.44493

Figure 4.5, Figure 4.6, Figure 4.7, and Figure 4.8 show the comparisons between the training data and the reconstructed data for each variable.

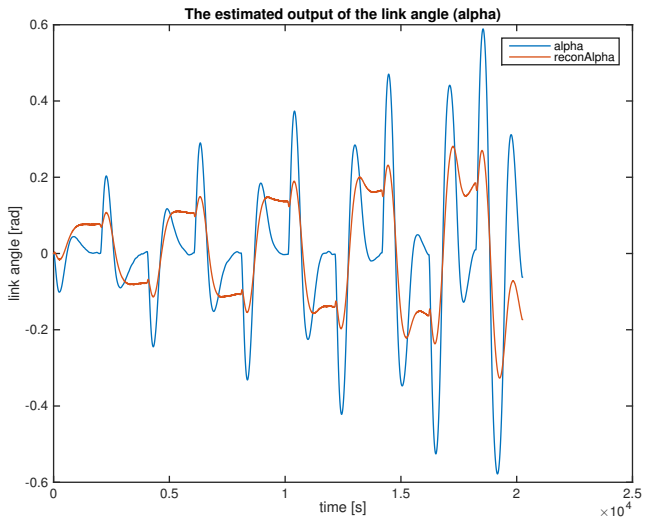


Figure 4.5: *link angle*

Figure 4.7, Figure 4.8, and Figure 4.6 indicate the reconstructed data match to the training data. Especially the wheel angle and angular velocity, which almost generate zero errors. Figure 4.5 shows that the reconstructed data of the link angle does not match their training data well. It is the main term that generates a mean square error of 0.36524. This is because the link is unstable over the wheel, but it is controllable. From the reconstructed figure, we can also see that the link angle changes regularly. This may make it possible to control it by using the reduced information.

The reduced three-dimensional representations are presented in Figure 4.9,Figure 4.10, and Figure 4.11.

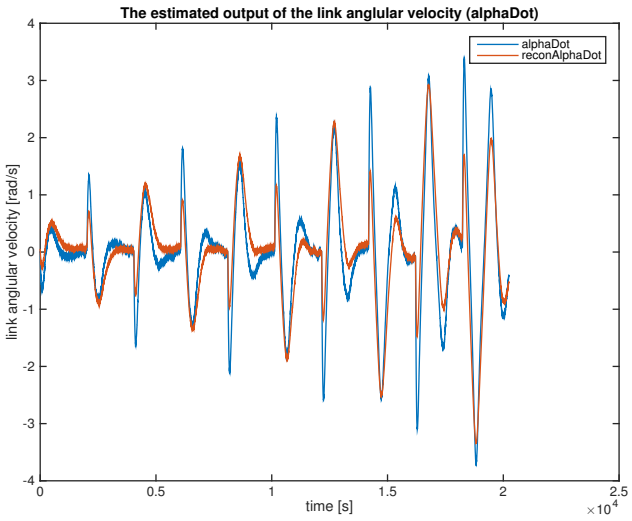


Figure 4.6: *link anglular velocity*

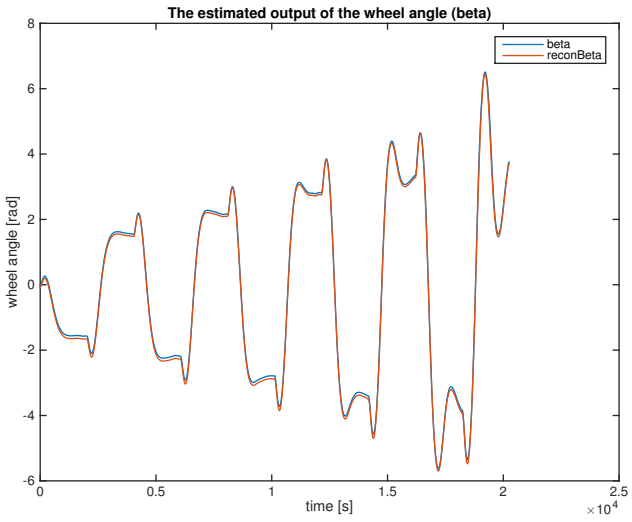


Figure 4.7: *wheel angle*

4.2. VALIDATION

In order to validate the model of the stacked auto-encoder obtained above, a chirp signal reference is provided for the same system. Run the same model, the comparisons between the reconstructed data and the training data are presented in Figure 4.12, Figure 4.13, Figure 4.14, and Figure 4.15. These figures demonstrate that the angle and angular velocity of the wheel match to each other perfectly. However, the information relates to the link do not match to each other well. This is corresponding to the learning results above. Thus, we have to conclude that when the system states pass through the stacked auto-encoder, they must be loosing some information. Whether the loosed

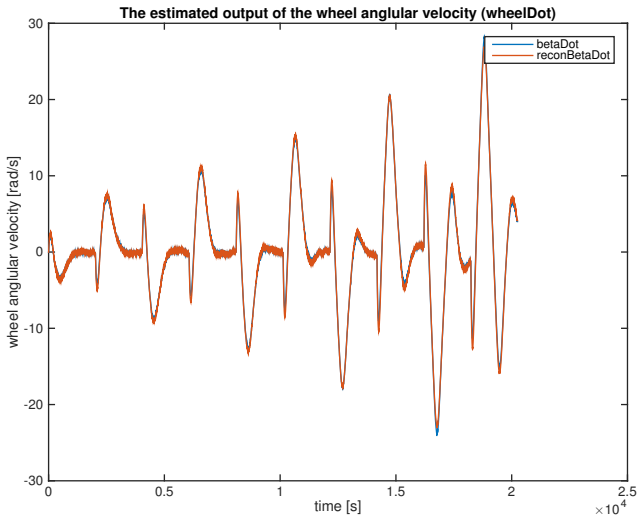


Figure 4.8: *wheel angular velocity*

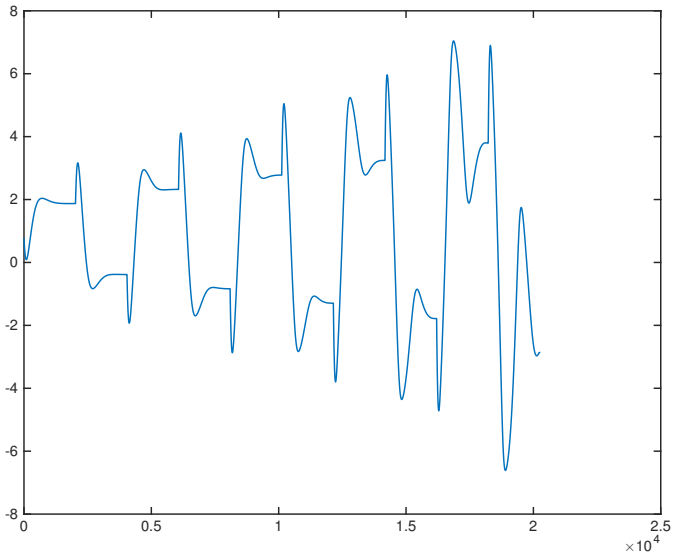


Figure 4.9: *mapped variable 1*

information is important or not for our control goal can not be decided at present. It will be verified in controlling the segway system in chapter 5.

4.3. CONCLUSION

In this chapter, the SAE experiment is implemented in Section 4.1. It decides the values of all the parameters and provides training results with those parameters values. Then the SAE mode is validated in Section 4.2. From these results, we can see that three out of four variables can obtain good

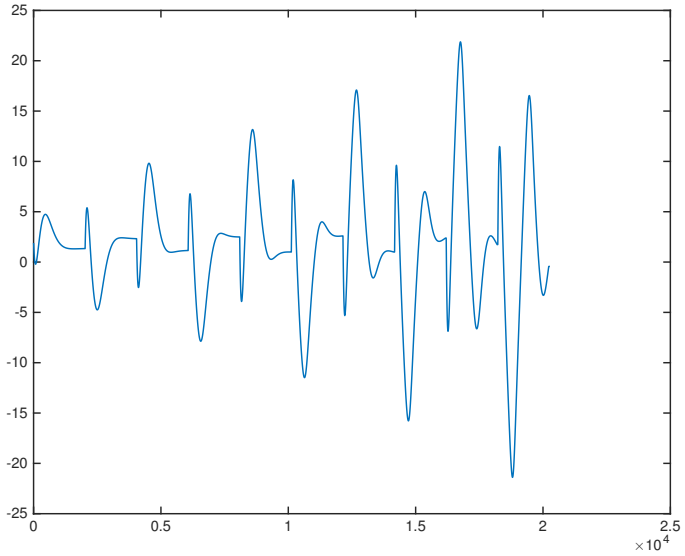


Figure 4.10: *mapped variable 2*

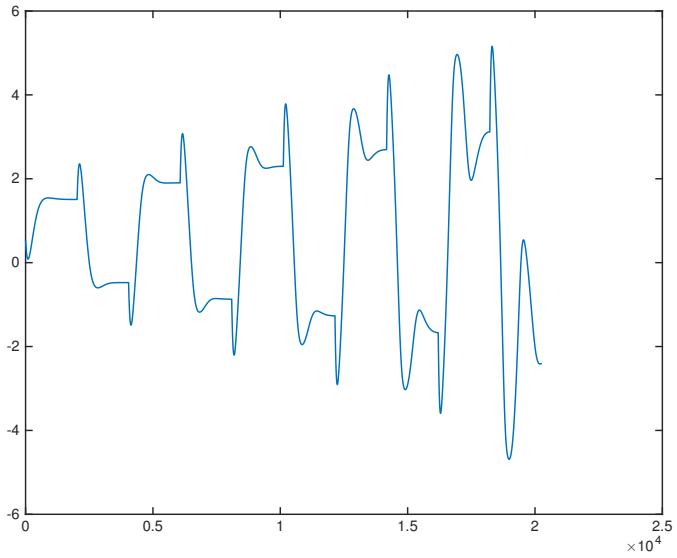


Figure 4.11: *mapped variable 3*

reconstructed results from the stacked auto-encoder. However, for variable link angle, the reconstructed data does not match the training data well. It can be improved by increasing the number of the first hidden layer units. Nonetheless, since the limitation of computer capability, 5000 is large enough . In addition, the time costing is also an important factor in practice experiment. In our experiment, it needs about one day to finish training the stacked auto-encoder once, which is a long time. Thus, considering these two factors, the limitation of the first hidden layer is set as 5000 in

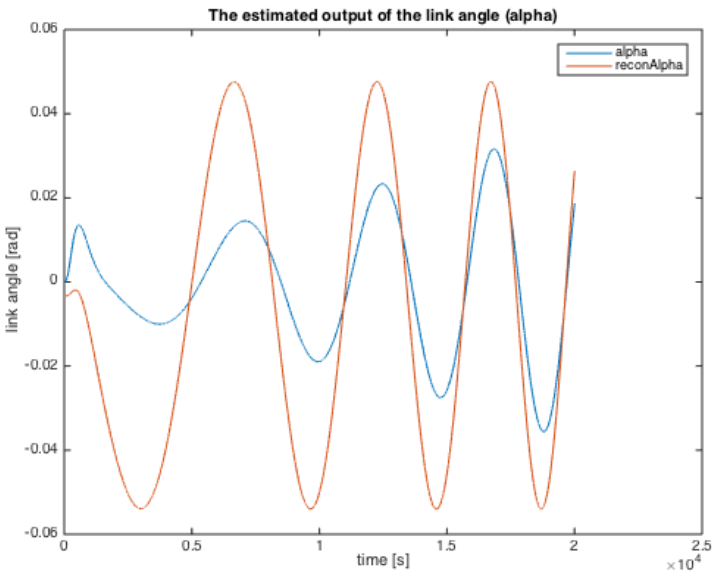


Figure 4.12: *link angle*

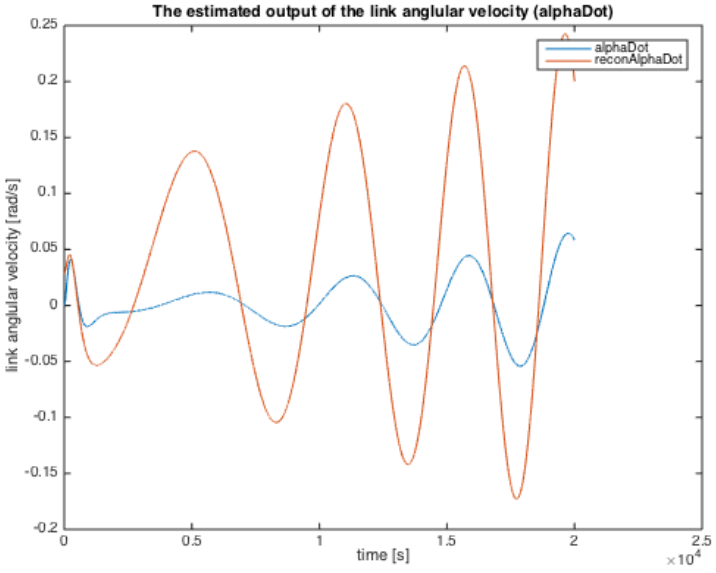


Figure 4.13: *link anglular velocity*

this project, even though the reconstructed alpha does not match well to the training data. In order to check whether important information relates to the control goal is missed or not, this stacked auto-encoder will be used in the experiment described in chapter 5.

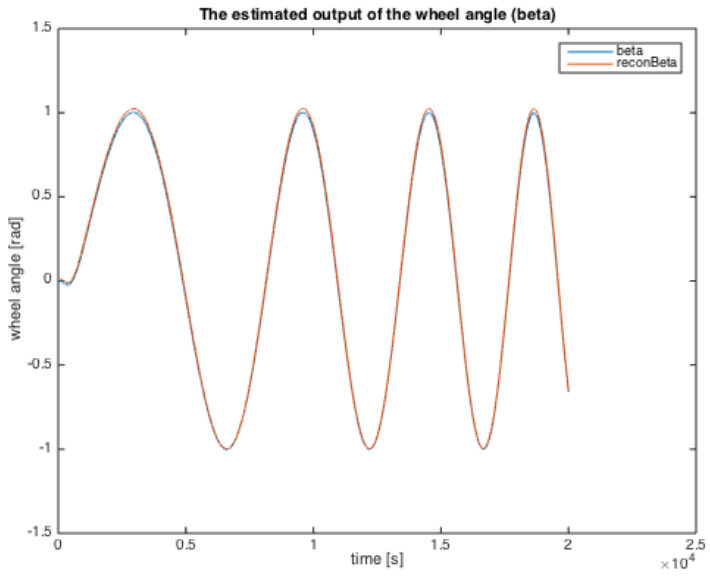


Figure 4.14: *wheel angle*

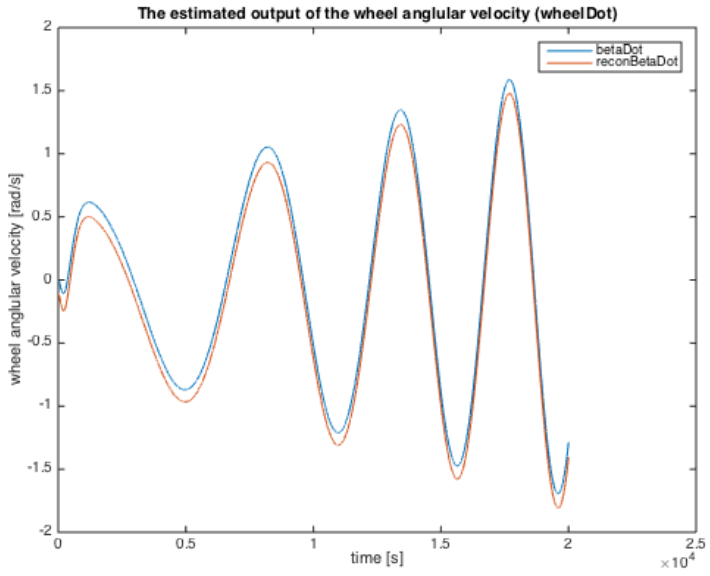


Figure 4.15: *wheel angular velocity*

5

DEEP REINFORCEMENT LEARNING IN CONTROL

In this chapter, the model obtained from chapter 4 will be integrated into reinforcement learning in chapter 3 such that control the four-dimensional Segway system with only three-dimensional representation. Section 5.1 introduces the deep reinforcement learning algorithm, and Section 5.2 provides the controlling result of the algorithm.

5.1. DRL ALGORITHM

Take the standard actor-critic method and above stacked auto-encoder as an example, the Deep Reinforcement Learning (DRL) algorithm that will be used in this thesis is implemented in Algorithm 3. When giving an action to the system, the system output state is transferred to the stacked auto-encoder first, and then the stacked auto-encoder gives a low-dimensional representation, which will be used to compute the value function and the policy. Finally, the policy generates a new action for the system. This process is repeated until achieve its control goal.

5.2. DRL EXPERIMENT

5.2.1. SELECTIONS OF PARAMETERS

In order to compare the performance between the deep reinforcement learning and the actor-critic method in Subsection 3.1.2, the same parameter values given in Table 3.3 and the weighted matrix

Algorithm 3 Actor-critic algorithm

Require: : the trace decay rate λ, the discount factor γ, critic learning rate α_c, and actor learning rate α_a.

initialize $e_0 = 0, \forall x$, and initialize θ_0, and ϑ_0.

Apply random input u_0

Start from $k \leftarrow 1$

loop

randomly choose Δu_k

Measure x_k, r_k

$u_k \leftarrow \pi(x_k, \vartheta_{k-1}) + \Delta u_k$

Apply u_k, obtain state $x_k^{'}$

Run $x_k^{'}$ through deep auto-encoder, obtain x_k

$\delta_k = r_k + \gamma V(x_k, \theta_{k-1}) - V(x_{k-1}, \theta_{k-1})$

$$e_k(x) = \begin{cases} 1, & \text{if } x = x_k \\ \lambda\gamma e_{k-1}(x) & \text{otherwise} \end{cases}$$

$\theta_k = \theta_{k-1} + \alpha_c \delta_k \sum_{x_v \in X_v} \frac{\partial V(x,\theta)}{\partial \theta}|_{x=x_v, \theta=\theta_{k-1}} e_k(x_v)$

$\vartheta_k = \vartheta_{k-1} + \alpha_a \delta_k \Delta u_{k-1} \frac{\partial \pi(x,\vartheta)}{\partial \vartheta}|_{x=x_{k-1}, \vartheta=\vartheta_{k-1}}$

update k as $k \leftarrow k+1$

end loop

P is applied in this experiment, parameter Q is changed to Equation 5.2.2, which adds one more parameter in Q since the join of wheel angular velocity $\dot{\beta}$. Since $\dot{\beta}$ is not so important in practice, so it is defined as 0 here. The parameter B should be re-defined by checking their RBFs plots according to the ranges of the reduced representations. Suppose vector $[\hat{\chi}_1 \quad \hat{\chi}_2 \quad \hat{\chi}_3]$ indicates the reduced representation of system state. Then the range of these variables are:

$$\hat{\chi}_1 \in [-6.6127,\ 7.0411], \qquad \hat{\chi}_2 \in [-21.3849,\ 21.8703], \qquad \hat{\chi}_3 \in [-4.6885,\ 5.1585] \tag{5.2.1}$$

According to these ranges, RBFs plots for each two of these variables $[\hat{\chi}_1,\ \hat{\chi}_2]$, $[\hat{\chi}_2,\ \hat{\chi}_3]$, and $[\hat{\chi}_1,\ \hat{\chi}_3]$ are shown in Figure 5.1, Figure 5.2, and Figure 5.3, with matrix B given in Equation 5.2.3.

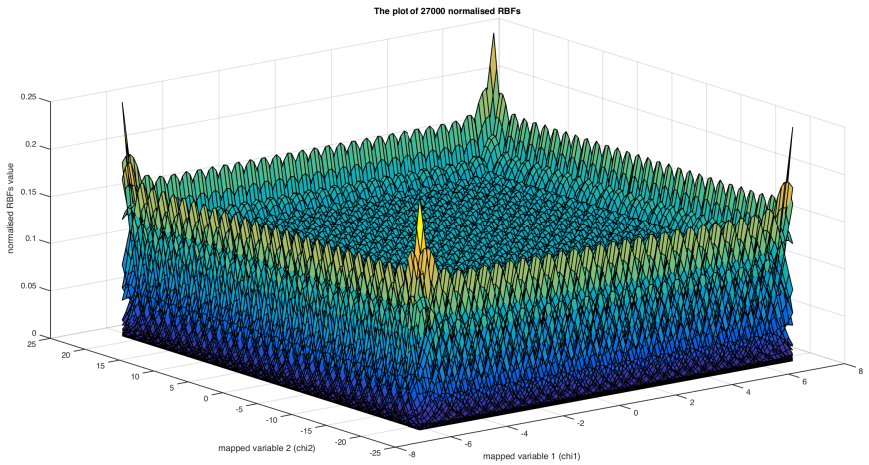


Figure 5.1: *RBFs plot for* $\hat{\chi}_1$ *and*$\hat{\chi}_2$

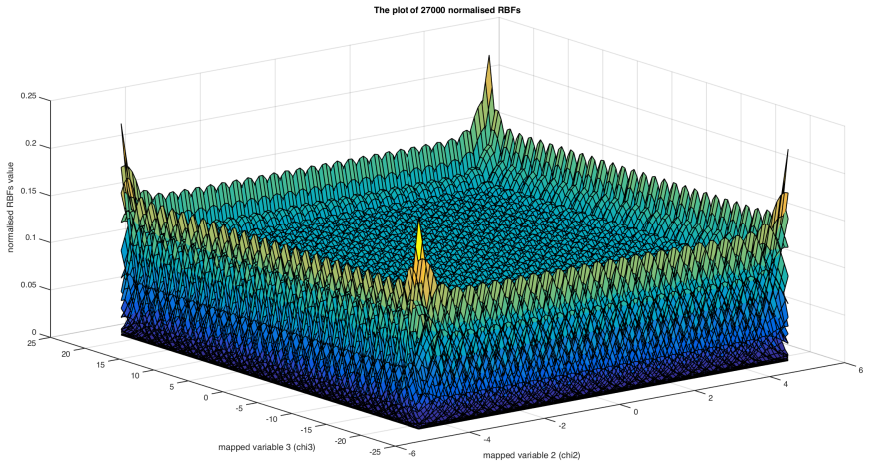


Figure 5.2: *RBFs plot for* $\hat{\chi}_2$ *and*$\hat{\chi}_3$

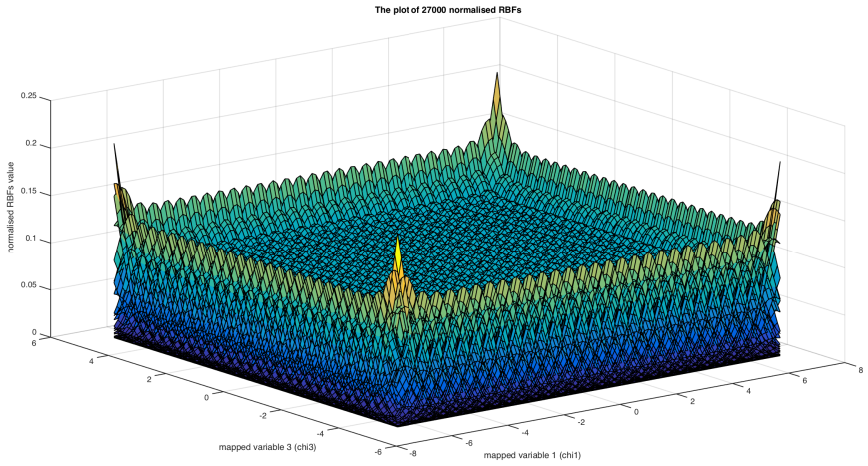


Figure 5.3: *RBFs plot for* $\hat{\chi}_1$ *and*$\hat{\chi}_3$

$$Q = \begin{bmatrix} 2.8 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 2.7 \end{bmatrix} \tag{5.2.2}$$

$$B = \begin{bmatrix} 0.35 & 0 & 0 \\ 0 & 2.8 & 0 \\ 0 & 0 & 0.25 \end{bmatrix} \tag{5.2.3}$$

5.2.2. EXPERIMENT RESULTS

Giving an initial state $[-\pi/8\ 0\ 0\ 0]$ and a random input $u \in [-5,5]$, the DRL starts to learn to control the segway system within 500 trials. Since the join of stacked auto-encoder, it needs $5.3s$ to run each trial, which is a little bit longer than in the experiment of Subsection 3.1.2. The learning result is given in Figure 5.4. It indicates the system is not successfully controlled within 500 trials. The sum of rewards is not apparently increased during the learning process, and the temporal difference does not converge to zero at the last trial. Figure 5.5 indicates the final state of the link is fell down in the vertical direction and stop at that position. Thus, we conclude that with the same parameter values and limited trail number, the control task is failed.

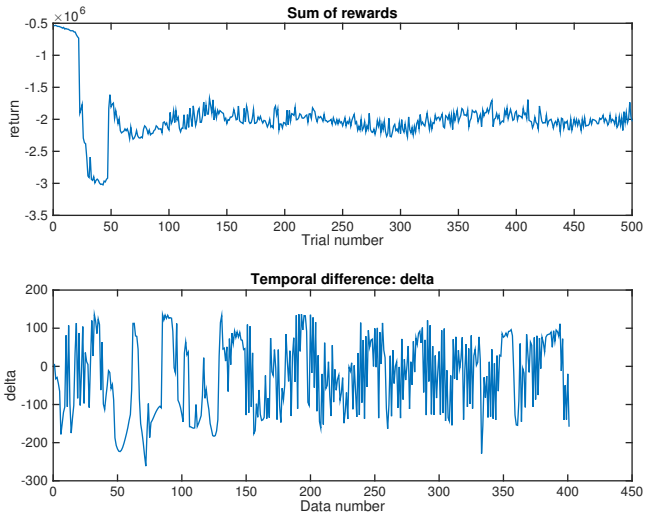


Figure 5.4: *Results of reward sums and temporal differences for the last trial*

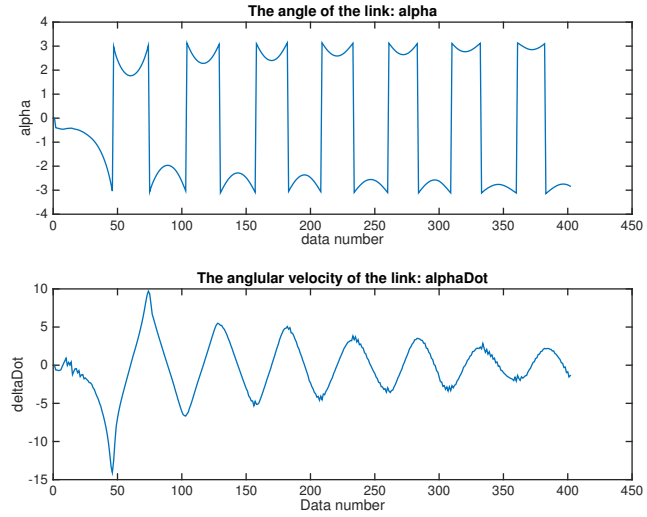


Figure 5.5: *State of link angle and angular velocity for the last trial*

5.3. DISCUSSION

In order to find out the problem that leads to the unsuccessful controlling, the individual effect of input and state on reward is given in Figure 5.6. From this figure, we can see that the sum of reward is mostly affected by the input, while the state seems like no change all the time. In order to increase the effect of the state, weight matrix Q needs to be improved.

With a new matrix (Equation 5.3.4), we have the new learning result presented in Figure 5.7 after 5000 trials. From the sum of the reward figure illustrates that the reward sum does not descend to zero but increasing in general. Figure 4.5 gives an explanation. It indicates that the reconstructed

data follows from the training data but cannot reach the same height. Take 0.5 in training data set as an example, the reward generated by this point is $r = -0.5^2 * Q_{11}$, with Q_{11} indicateing the weight for the first variable. However, when this point through the stacked auto-encoder, it becomes $r = -0.2^2 * Q_{11}$, which has decreased in comparison. This results in the learning rate decreasing during the each trial. As a result, this slows the entire learning process. We conclude that with more trial numbers, it should achieve our control goal. Furthermore, (Kaelbling *et al.*, 1996) proposes that quick learning is one of the criterion for reinforcement learning's successful application in practice. That is to say the number of trials needed and the time spent for each trial should be limited. In our experiment, 5000 is large enough, even though it can balance the link with more trials, it contradicts with our purpose to improve the efficiency of RL. We conclude that some information was lost when states pass through the stacked auto-encoder. It is sufficient to conclude that if it cannot control the segway system after 5000 trials, the control task is unsuccessful.

$$Q = \begin{bmatrix} 12.5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0.1 & 0 \\ 0 & 0 & 0 & 0.01 \end{bmatrix} \tag{5.3.4}$$

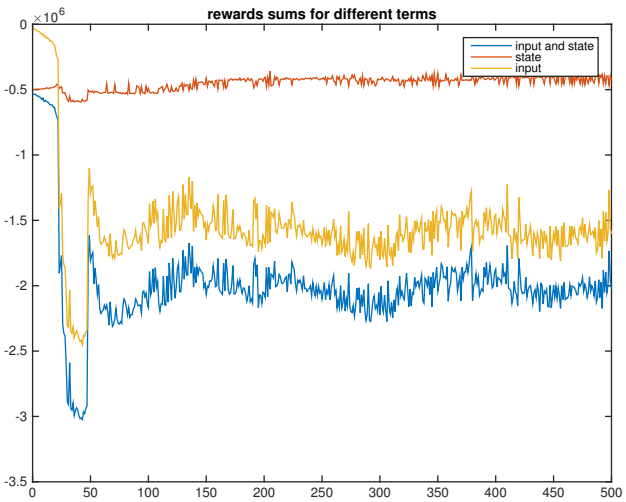


Figure 5.6: *individual effect of state and input on reward*

This situation cannot be improved by increasing the value of Q because the system becomes unstable when Q is set to a large value. To make it stable again, many other parameters need to be tuned. Such as the range of each variable and the value of the matrix B. Due to the lack of knowledge of the physical meaning of the reduced representation, it is difficult to adjust theses parameters

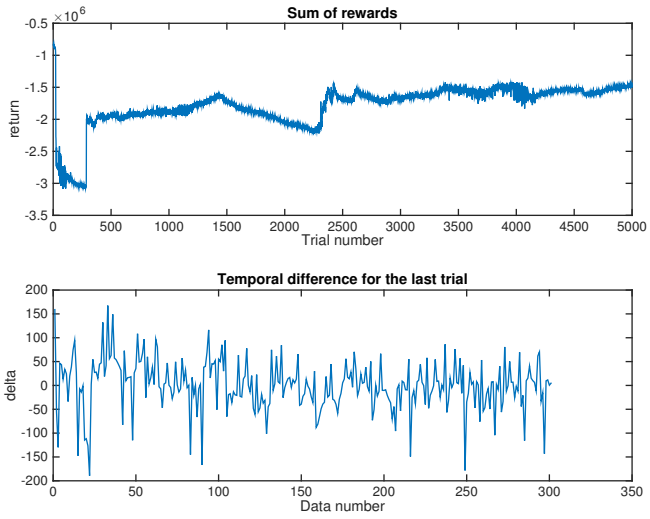


Figure 5.7: *sum of rewards and temporal difference with new Q*

properly. For instance, in the system above, the range of the link angle α must be within $[-\pi,\ \pi]$. But when this data passes through the stacked auto-encoder, it is difficult to obtain its new range. Without this range, it is impossible to get the new value of B.

This chapter describes the deep reinforcement learning algorithm introduced in Section 3.3. Results from this experiment confirmed that some important information was lost in the reduced representation in Subsection 4.1.4.

6

Conclusion and Future Work

This thesis presented a first attempt integrating deep neural networks into reinforcement learning by means of estimating the value functions and auto-encoders for system states. Although the segway control experiment showed that this new method still requires further optimisation, there are still many lessons learnt and possibilities for future improvement. In this thesis, we first verified the standard actor-critic method for segway control. Different from these standard methods, my work demonstrated how reinforcement learning can be used to control a system with its sub-state but not the full state. This was achieved by controlling the segway with both sub-states used aiming at the same control goal. My work is a first attempt to reduce the dimension of system state space using stacked auto-encoder. Despite the impact of the unstable property, the reconstructed data matches the training data perfectly. This project is also another apply the theory of reinforcement learning to the implementation of control systems, especially robots.

Reinforcement learning algorithms have been well used in control systems. Many parameters involved in reinforcement learning control are tuned based on experience locally. This results in some locally optimal parameters but the combination may not be optimal at system level. Learning performance might be improved by tuning some of these parameters using some multi-objective algorithms. Therefore some more systematic methods for deciding parameter combination are desired. Combining deep neural networks with reinforcement learning could also be interesting. For example, neural networks have been used to construct nonlinear observer in (Ahmed and Riyaz, 2000) and (Grondman *et al.*, 2012b). Taking advantage of this, the implemented observers are ex-

pected to be more dynamic in systems. Following the work of stacked auto-encoder, it was also noticed that a probabilistic graphical model, namely the Deep Boltzmann Machine (DBM) (Salakhutdinov and Hinton, 2012), can possibly also be used in control. Even though the control of segways has it limitations, this new method provides a new bridge connecting reinforcement learning and deep neural networks. This could also be a meaningful case study inspiring applying deep neural networks on other problems.

BIBLIOGRAPHY

R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*, Vol. 1 (MIT press Cambridge, 1998).

G. E. Hinton and R. R. Salakhutdinov, *Reducing the dimensionality of data with neural networks,* Science **313**, 504 (2006).

K.-E. Arsén, *Knowledge-based control systems,* in *American Control Conference, 1990* (IEEE, 1990) pp. 1986–1991.

I. Grondman, L. Buşoniu, G. A. Lopes, and R. Babuška, *A survey of actor-critic reinforcement learning: Standard and natural policy gradients,* Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on **42**, 1291 (2012a).

S. J. Bradtke, B. E. Ydstie, and A. G. Barto, *Adaptive linear quadratic control using policy iteration,* in *American Control Conference, 1994*, Vol. 3 (IEEE, 1994) pp. 3475–3479.

C. J. Watkins and P. Dayan, *Q-learning,* Machine learning **8**, 279 (1992).

G. A. Rummery and M. Niranjan, *On-line q-learning using connectionist systems,* (1994).

R. S. Sutton, *Learning to predict by the methods of temporal differences,* Machine learning **3**, 9 (1988).

V. Aleksandrov, V. Sysoyev, and V. Shemeneva, *Stochastic optimization*, Vol. 5 (1968).

P. W. Glynn, *Likelilood ratio gradient estimation: an overview,* in *Proceedings of the 19th conference on Winter simulation* (ACM, 1987) pp. 366–375.

R. S. Sutton, D. A. McAllester, S. P. Singh, Y. Mansour, *et al.*, *Policy gradient methods for reinforcement learning with function approximation.* in *NIPS*, Vol. 99 (Citeseer, 1999) pp. 1057–1063.

J. Peters, K. Mülling, and Y. Altun, *Relative entropy policy search.* in *AAAI* (2010).

I. H. Witten, *An adaptive optimal controller for discrete-time markov environments,* Information and control **34**, 286 (1977).

I. Grondman, M. Vaandrager, L. Busoniu, R. Babuska, and E. Schuitema, *Efficient model learning methods for actor–critic control,* Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on **42**, 591 (2012b).

W. S. McCulloch and W. Pitts, *A logical calculus of the ideas immanent in nervous activity,* The bulletin of mathematical biophysics **5**, 115 (1943).

S.-T. Hu, *Threshold logic* (Univ of California Press, 1965).

F. Rosenblatt, *The perceptron: a probabilistic model for information storage and organization in the brain.* Psychological review **65**, 386 (1958).

T. Nakanishi, *Machine learning algorithms applied to financial forecasting,* .

Y. Bengio, I. J. Goodfellow, and A. Courville, *Deep learning,* (2015), book in preparation for MIT Press.

D. E. Rumelhart, G. E. Hinton, and R. J. Williams, *Learning internal representations by error propagation*, Tech. Rep. (DTIC Document, 1985).

A. Ng, *Sparse autoencoder,* CS294A Lecture notes **72** (2011).

P. J. Werbos, *The roots of backpropagation: from ordered derivatives to neural networks and political forecasting*, Vol. 1 (John Wiley & Sons, 1994).

D. E. Rumelhart, G. E. Hinton, and R. J. Williams, *Learning representations by back-propagating errors,* Cognitive modeling **5**, 3 (1988).

Deep networks:overview, UFLDL Tutorial .

E. Najafi, R. Babuska, and G. A. Lopes, *An application of sequential composition control to cooperative systems,* in *Robot Motion and Control (RoMoCo), 2015 10th International Workshop on* (IEEE, 2015) pp. 15–20.

J. Kober, J. A. Bagnell, and J. Peters, *Reinforcement learning in robotics: A survey,* The International Journal of Robotics Research , 0278364913495721 (2013).

W. B. Powell, *AI, OR and control theory: A rosetta stone for stochastic optimization*, Tech. Rep. (Technical report, Princeton University, 2012).

J. Nakanishi, R. Cory, M. Mistry, J. Peters, and S. Schaal, *Operational space control: A theoretical and empirical comparison,* The International Journal of Robotics Research **27**, 737 (2008).

S. Schaal, C. G. Atkeson, and S. Vijayakumar, *Scalable techniques from nonparametric statistics for real time robot learning,* Applied Intelligence **17**, 49 (2002).

L. Van der Maaten, E. Postma, and H. van den Herik, *Matlab toolbox for dimensionality reduction,* MICC, Maastricht University (2007).

L. P. Kaelbling, M. L. Littman, and A. W. Moore, *Reinforcement learning: A survey,* Journal of artificial intelligence research , 237 (1996).

M. Ahmed and S. Riyaz, *Design of dynamic neural observers,* IEE Proceedings-Control Theory and Applications **147**, 257 (2000).

R. Salakhutdinov and G. Hinton, *An efficient learning procedure for deep boltzmann machines,* Neural computation **24**, 1967 (2012).

A

GLOSSARY

The glossary consists of a summary of the mathematical symbols used throughout this thesis.

A.1. MDPs AND RL

Symbols	Meanings
s	current state of a system
S	a set of all possible states
r	a reward
R	a set of all the rewards
u	an action
A	a set of actions
s_k	state k
r_k	reward k
u_k	action k
f	state transition function
ρ	reward function
γ	discount factor
π	policy
δ	return error

A.1.1. APPROXIMATE RL

Symbols	Meanings
ϑ	poicy parameter vector
π_ϑ	parameterized policy
$V(x,\theta)$	approximate value function
ϕ	basis function
α_c	critic learning rate
θ	critic parameter vector
e_k	eligibility trace
λ	trace decay rate
X_v	all visited states in one trial
Δu_k	zero-mean random exploration
α_a	actor learning rate
Φ_s	Gaussian response base on distance
c_i	center state of feature i
σ	feature's width
$\phi(x)$	normalised RBFs
B	width of RBFs

A.1.2. SEGWAY MODEL

Symbols	Meanings
$M(q)$	inertia matrix
$C(q,\dot{q})$	Coriolis and centrifugal forces
$G(q)$	Gravity matrix
$B(q)$	input vector
F_{ext}	external forces
I_b	Link inertia
m_b	Link mass
I_W	Wheel inertia
m_W	Wheel mass
g	Gravity
l	Link half length
r	Wheel radius
b_g	Rolling friction
b_j	Joint friction
K_t	Torque constant
k_e	Back EMF
R_m	Rotor resistance
N_g	Gear ratio
N_s	Input gain

A.2. NEURAL NETWORK

Symbols	Meanings
$f(\cdot)$	activation function
i	i^{th} input of NN
ω_i	weight of i^{th} input
L_i	i^{th} layer of NN
$x_i^{l_j}$	i^th input of layer l_j
b^{l_j}	bias in layer l_j
$\omega_{ij}^{l_{12}}$	weight between i^th node in layer l_1 and j^th node in layer l_1
y_j^{l+1}	hidden layer output
$h_{W,b}(x)$	final layer output
$J(W,b)$	cost function

B

ACRONYM

The glossary consists of a summary of the abbreviations used throughout this thesis.

Abbreviations	Full names
RL	Reinforcement Learning
AC	Actor-Critic method
DNNs	Deep Neural Networks
AI	Artificial Intelligence
MDP	Markov Decision Process
TD	Temporal Difference
RBF	Radial Basis Function
LLR	Local Linear Rgression
ANN	Artificial Neural Networks
MLP	Multi-Layer Perceptron
SAE	Stacked Auto-Encoder
BP	Back Propagation
PID	Proportional-Integral-Derivative
LQG	Linear-Quadratic-Regulator
LSQL	Least-Squares Q-Learning