Department of Computer Science
Series of Publications A
Report A-2015-0

Deep Recurrent Neural Network for Robot Reinforcement Learning

Yuan Gao

University of Helsinki
Finland

Supervisor

Dorota Glowacka, University of Helsinki, Finland
Leo Kärkkäinen, Nokia Research Center, Finland
Honkala Mikko Nokia Research Center, Finland

Pre-examiners

Opponent

Custos

Contact information

Department of Computer Science
P.O. Box 68 (Gustaf Hällströmin katu 2b)
FI-00014 University of Helsinki
Finland

Email address: info@cs.helsinki.fi
URL: http://cs.helsinki.fi/
Telephone: +358 2941 911, telefax: +358 9 876 4314

ISSN 1238-8645
ISBN 000-00-0000-0 (paperback)
ISBN 000-00-0000-0 (PDF)
Computing Reviews (1998) Classification: A.0, C.0.0
Helsinki 2015
Unigrafia

Deep Recurrent Neural Network for Robot Reinforcement Learning

Yuan Gao

Department of Computer Science
P.O. Box 68, FI-00014 University of Helsinki, Finland
gaoyuankidult@gmail.com
http://www.cs.helsinki.fi/u/yuangao/

PhD Thesis, Series of Publications A, Report A-2015-0
Helsinki, September 2015, 7 pages
ISSN 1238-8645
ISBN 000-00-0000-0 (paperback)
ISBN 000-00-0000-0 (PDF)

Abstract

Computing Reviews (1998) Categories and Subject Descriptors:
A.0 Example Category
C.0.0 Another Example

General Terms:

Additional Key Words and Phrases:

Acknowledgements

Contents

Chapter 1

Introduction

Controlling a complicated mechanical system to perform a certain task, for example making robot to dance, is a traditional problem studied in the field of control theory. Many successful applications like Google BigDog[18] and Google Self-driving car [4] have been made in accordance to the new theories found in this field.

However more evidences show that in-cooperating with machine learning techniques in robotics can enable people to get rid of tedious engineering works of adjusting environmental parameters. Many researchers like Jan Peters, Sethu Vijayakumar, Stefan Schaal, Andrew Ng and Sebastian Thrun are the early explorers in this field. Based on the Partial Observable Markov Decision Process(POMDP) reinforcement learning, they contributed first several algorithms that enable robot to learn to perform a certain task overtime.

Recently, one sub-field of machine learning called deep learning gained a lot of attention as a method attempting to model high-level abstractions by using model architectures composed by multiple non-linear layers. (for example [10]). Several architectures of deep learning networks like deep belief network [5], deep Boltzman machine [19], convolutional neural network [10] and deep de-noising auto-encoder [24] have shown its advantages in specific areas. Especially, one of convolutional neural networks, which was invented by Krizhevsky, outperformed all the traditional feature-based machine learning techniques in ImageNet competition.

Based on the two trends we noticed, a natural path of research is to use deep learning methods for controlling movements of robot. Until the end of 2014, the main works of deep learning are more related to a category of robotics called perception, which deals with problems like Sensor Fusion [16], Nature Language Processing(NLP)[1] and Object Recognition[11][7]. Although considered briefly in Jürgen Schmidhuber's team[12], the other

area of robotics, namely control, remains more-or-less unexplored in the realm of deep learning.

The researches done in Jürgen Schmidhuber's team provided several interesting structures that might be potentially useful for robot control. The name of one of these structures is called Long Short Term Memory (LSTM), which is one variation of Recurrent Neural Network(RNN). Several experiments like generating sequences[2], speech recognition[**?**] and neural turing machine [3] show that it has ability of extracting and storing temporal information from data. As a consequence, this specific structure of RNN, with modification, can be applied for control problems of robots.

Currently the main machine learning algorithm used for learning a action is based on reinforcement learning. Specifically, it is based on a category called policy gradient algorithm, which means the algorithm needs to directly search in policy space instead of state-action space. In this framework, there are normally two parts, namely actor part and critic part. Actor part is used for generating different actions based on policy parameters, the critic part is for simulating the environment in order to provide an accurate approximation for the environment. Deep Learning can be used in this case as it is general function approximator and environment is considered as a function that takes several parameters in and output an action.

In thesis, we consider several kind of deep learning models to approximate the environment to reduce the real word samples needed. As a consequence, two major aspects are considered, one main focus of this Thesis is to introduce general learning methods for robot control problem with an emphasize on deep learning method. this thesis tries to describe the transitional learning method as well as the emerging deep learning methods for robot control problem. Another focus of this thesis is to introduce the main contribution of author in this field. With experiments, the author is able to show his own method can outperform the transitional machine learning methods of robot control problems.

Chapter 2

Reinforcement Learning

If we would like to discuss what might be the most common way of learning, learning based on interacting with our environment is a natural idea to think about. When we were born in this world, we had no teachers around us. But tens of years passed, we learned to fear, to communicate with others and to write a paper. As a consequence, it is very natural to think that our environment is a great source of information. While playing around with environment, we learn by taking actions and getting reward from it. Now when we cook , when we do exercise, we are fully and acutely aware what will be the response of environment.

The RL is an area that studies the mechanism of the such kind of learning in a computational way. Generally the goal of RL is to find a way of mapping different states with different actions so that we could maximize the reward signals.

There are two main approaches in area of reinforcement learning, one is based on Markov Decision Process(MDP)[**?**]. Both methods have advantages and drawbacks when applied to robotics, another one is recurrent neural network(RNN)[**?**].

In the following sections of this chapter, we may consider robot as an agent in all description of related techniques.

2.1 Markov Decision Process

Markov Decision Process (MDP) is a discrete time stochastic control process. We may consider a robot in a state s of discrete state space S. The robot can take an action a in all possible action set A resulting in a state s' . We can denote this process as transition function $P_a(s, s')$ meaning the probability of moving from state s to state s' through action a. Then after

the robot executes action a and results in s' , it will receive a reward r according to reward function denoted as $R_a(s, s')$. The goal of reinforcement learning is to optimize cumulative reward of whole process.

The problems of MDP is more clear to researchers as it is based on mathematical formalizations. On one hand, MDP-based methods together with optimization methods such as Gradient Partially Observable Markov Decision Processes (GPOMDP), projection method or nature gradient are state-of-art in robot trajectory learning. On another hand, as data collected from robot is different from normal data, it was pointed out the MDP-based methods suffers several curses[9].

- Curse of Dimensionality
- Curse of Real-World Samples
- Curse of Under-Modeling and Model Uncertainty
- Curse of Goal Specification

The data is normally high dimensional, continuous and erroneous data in robot systems. It is considered to be difficult questions to neglect these issues and it is also hard to specify the goal of system i.e. what robot needs to be at last.

2.1.1 Partially Observable Markov Decision Process

A partially observable Markov decision process (POMDP) is a generalization of a Markov decision process (MDP). A POMDP models an agent decision process in which it is assumed that the system dynamics are determined by an MDP, but the agent cannot directly observe the underlying state. Instead, it must maintain a probability distribution over the set of possible states, based on a set of observations and observation probabilities, and the underlying MDP.

The POMDP framework is general enough to model a variety of real-world sequential decision processes. Applications include robot navigation problems, machine maintenance, and planning under uncertainty in general. The framework originated in the operations research community, and was later taken over by the artificial intelligence and automated planning communities.

An exact solution to a POMDP yields the optimal action for each possible belief over the world states. The optimal action maximizes (or minimizes) the expected reward (or cost) of the agent over a possibly infinite

horizon. The sequence of optimal actions is known as the optimal policy of the agent for interacting with its environment.

More precisely, the POMDP is a discrete time stochastic control process. We may consider a robot in a state s_t of discrete space $\mathcal{S}$ where t means iteration number. Now the robot can take an action a in all possible action set A resulting in a state s_{t+1}. We can denote this process as transition function Ta(s_t,s_{t+1}) meaning the probability of moving from state s_t to state s_{t+1} through action a. Then after the robot executes action a and results in st+1, it will receive a reward according to reward function denoted as $R_a(s_t, s_{t+1})$.

Formally, we define MDP as follows:

A Markov decision process is a 4-tuple denoted as ($\mathbb{S}$,$\mathbb{A}$,$T \cdot (\cdot, \cdot)$,$R \cdot (\cdot, \cdot)$) In this tuple

- $\mathbb{S}$ is a finite set of states. It describes all possible states of the space.
- $\mathbb{A}$ is a finite set of actions. It describes all possible states that robot can take.
- $T \cdot (\cdot, \cdot)$ is a function that takes three arguments. e.g $T_{a_t}(s_t, s_{t+1})$ means probability of transferring from s_t to s_{t+1} with action a_t.
- $R \cdot (\cdot, \cdot)$ is also a function that takes three arguments. e.g $R_{a_t}(s_t, s_{t+1})$ means reward of transferring from s_t to s_{t+1} with action a_t.

The main problem of reinforcement learning is to find a policy function $\pi(s) : s \to a$ to map every state s with action a so that a cumulative reward R is maximized, where $0 \leq \gamma \leq 1$ is a discount factor. As the discount factor generalizes discounted situation and undiscounted situation, it broadens our theory.

2.1.2 Markov Decision Process with Continuous States

Now considering the environment of robots, we need some modifications to original POMDP. First we still assume the process to be discrete time but we consider continuous space $\mathbb{S} \subseteq R^n$ and continuous action set $\mathbb{A} \subseteq R^m$ where n is dimension of space and m is dimension of actions. For initial state s_0, we assign a distribution $p(s_0)$ where $s_0 \in \mathcal{S}$. At any state $s_t \in \mathcal{S}$, we have a continuous policy $\pi(a_t|s_t) = p(a_t|s_t, \theta)$ parametrized by θ. Transfer function now also becomes continuous. It is corresponded to a probability distribution $T_{a_t}(s_t, s_{t+1}) = p(s_{t+1}|s_t, a_t)$. After this step is completed, the process will general a reward function $R_{a_t}(s_t, s_{t+1})$ which is

defined as $R \cdot (\cdot, \cdot) : S \times S \times A \to [0, \text{inf})$. After these modifications, we can now formalize continuous MDP.

Continuous Markov Decision Process With Infinite States is a modified version of ordinary POMDP. Mathematically, it is defined as a 5-tuple $(P_{init}, S, A, T \cdot (\cdot, \cdot), R \cdot (\cdot, \cdot))$ where in this tuple

- P_{init} is a initial distribution of the states.
- $\mathbb{S} \subseteq \mathbb{R}^n$ is a infinite set of states. It describes all possible states of the space.
- $\mathbb{A} \subseteq \mathbb{R}^m$ is a infinite set of actions. It describes all possible actions of the agent.
- $T \cdot (\cdot, \cdot)$ is a function that takes three arguments. e.g $T_{a_t}(s_t, s_{t+1})$ means probability of transferring from st to st+1 with action at.
- $R \cdot (\cdot, \cdot)$ is a function that takes three arguments. e.g $R_{a_t}(s_t, s_{t+1})$ means reward of transferring from s_t to s_{t+1} with action at.

With this continuous setting, we have objective function defined as follows:

$$J(\theta) = \vec{E_\tau}\{(1-\gamma)\sum_{t=0}^{\infty}\gamma^t R_t|\theta\} \tag{2.1}$$

$$= \int_{\mathbb{S}} d^\pi(\vec{s}) \int_{\mathbb{A}} \pi(\vec{a}|\vec{s}) R(\vec{a}, \vec{s}) d\vec{a} d\vec{s} \tag{2.2}$$

Where $d^\pi(\vec{s})$ refers to $1 - \gamma^t p(\vec{s} = \vec{s_t})$, $0 \geq \gamma \geq 1$ refers to discount factor and $\pi(\vec{a}|\vec{s})$ is parametrized by θ.

2.1.3 Value Functions

We define two functions to further describe this process. First function is statevalue function denoted as $V^\pi(\vec{x})$. This means expected value of an agent that follows an policy of π with initial value $\vec{s}$. It characterizes the rewards of following a policy π. Mathematically, it is defined as:

$$V^\pi(\vec{s}) = \vec{E}_\tau\{\sum_{t=0}^{\infty}\gamma^t r_t|\vec{s} = \vec{s_0}\} \tag{2.3}$$

where τ stands for trajectory of agent.

$$Q^\pi(\vec{s}, \vec{a}) = \vec{E}_\tau\{\sum_{t=0}^{\infty}\gamma^t r_t|\vec{s} = \vec{s_0}, \vec{a} = \vec{a_0}\} \tag{2.4}$$

State-value function only depends on the first state of agent. After that the system is governed by policy π. Another value function we need to defines is state-action value function.

2.1.4 Natural Actor Critic Model

The first thing of demonstrating Natural Actor-Critic is to explain the term. Natural stands for gradient framework called natural gradient while Actor-Critic means an iterative method of evaluating and improving objective function.

Robot learning is based on Markov Decision Process with discrete time and continuous states. The objective function is defined in section Markov Decision Process by Formula 1. A normal gradient of objective function is defined as :

$$\nabla J(\theta) = \int_{\mathbb{S}} d^{\pi}(\vec{s}) \int_{\mathbb{A}} \nabla_{\theta} \pi(\vec{a}, \vec{s}) R(\vec{a}, \vec{s}) d\vec{a} d\vec{s} \tag{2.5}$$

Please note θ is related to $\pi(\vec{a}|\vec{s})$ as it is defined by $\pi(\vec{a}|\vec{s}|\theta)$.

However, here we need to redefine this gradient as vanilla gradient. Pointed out and nicely presented by Shun-ichi Amari, another kind of gradient called natural gradient is more efficient in many machine learning application than vanilla gradient. Mathematically we define following formula as natural gradient:

$$\nabla J(\theta) = G^{-1} \nabla J(\theta) \tag{2.6}$$

Where G is fisher information metrix[]. Fisher information matrix is a matrix defined on Romanian space. This metric is interesting on several aspects. One of them is that it shows true direction of a function's steepest direction and on the contrary, the vanilla gradient does not. The long proof of previous statement is based by showing natural gradient descent method is fisher efficient. We recommend a further reading on Amari's paper[].

If we rewrite Formula 5 as:

$$\nabla_{\theta} J(\theta) = \int_{\mathbb{S}} d^{\pi}(\vec{s}) \int_{\mathbb{A}} \nabla_{\theta} \pi(\vec{a}, \vec{s}) (Q^{\pi}(\vec{a}, \vec{s}) - b^{\pi}(\vec{x})) d\vec{a} d\vec{s} \tag{2.7}$$

Where $Q^{\pi}(a, s)$ is state-action value function and $b^{\pi}(x)$ is kind of baseline. Two papers [] [] demonstrate why $R(s, a)$ is replace by $Q^{\pi}(a, s) - b^{\pi}(x)$. It can be shown that $R(s, a)$ is further approximated as $(\nabla_{\theta} \log \pi(a, s)^{\pi}(s))^{T} \vec{w}$ parametrized by $\vec{w}$.

2.2 Reinforcement Learning Methods

In this section, we introduce common ideas in robotics including temporal difference learning, Episodic learning and policy gradient method.

2.2.1 Policy Evaluation

Policy evaluation is a process that computes state-value function based on a policy π. Generally, there are three ways of doing policy evaluation.

The simple every-visit Monte-Carlo method used for evaluating policy is defined as:

$$V^{\pi}(\vec{s}) \leftarrow V^{\pi}(s_t) + \gamma(R(s_t) - V(s_t)) \tag{2.8}$$

where $V^{\pi}(s_t)$ is the state-value of state s_t, γ is discount factor and $R(s_t)$ is the reward of state s_t.

The most well-known one is based on dynamic programming. It computes the best state-value for each state and iteratively update the value to each possible state. This update process can be defined as:

$$V^{\pi}(\vec{s}) \leftarrow \vec{E}\{r_{s_1} + \gamma V(s_t)\} \tag{2.9}$$

Temporal Difference Learning is another important concept in the area of reinforcement learning. It uses temporal difference information to learn a value function. As a result, it does not need to know every state.(this issue will be address)

As we can see, the simple every-visit Monte-Carlo method only in-cooperate information of current state. However, considering future information may result a better converging rate.

Based on this idea, the simplest TD method(TD(0)) is developed and defined as:

$$V^{\pi}(\vec{s}) \leftarrow V^{\pi}(s_t) + \gamma(r_{t+1} + \gamma V(s_{t+1}) - V(s_t)) \tag{2.10}$$

We may notice that each method requires different kind of data to be used. The simple every-visit Monte-Carlo method only in-cooperate information of current state, so it can be updated just by using the information observed at each state. The TD(0) requires successive two states and dynamic programming requires information of every state.

We may further consider the differences between these three methods based on characteristics they have. Generally, bootstrapping means using estimate of successor state in reinforcement learning. In this case, we can classify these three methods as follows:

Algorithm 1: Tabular TD(0) policy evaluation algorithm

Require: Policy π Initialize $V(s)$ arbitrarily ;
while *until s is terminal(for each episode)* **do**
a $\leftarrow$ action given by π for state s;
take action a, observe reward r and next state s'
$V^{\pi}(\vec{s}) \leftarrow V^{\pi}(s_t) + \gamma(r_{t+1} + \gamma V(s_{t+1}) - V(s_t))$ $s \leftarrow s'$

- MC does not bootstrap, as it always uses current state for updating
- DP bootstraps, as it needs to calculate expectation of all the successive states.
- TD(0) bootstraps, as it needs to know the state-value function of next state.

If we consider whether they use sampling method for not, we also can classify them as following:

- MC samples
- DP does not sample
- TD samples

Now I will use TD(0) policy evaluation algorithm as example to show how is policy evaluated.

2.2.2 Policy Gradient Methods

The previous policy evaluation algorithms updates state-value for each state. However, there are also episodic algorithms to evaluate policy based on parameters. In the following text, we are going to introduce policy gradient method for robotics.

Reinforcement learning is probably the most general framework where such robot learning problems can be phrased. Despite the fact that many reinforcement learning algorithm fails to scale where robots have more degrees of freedom, policy gradient methods is one of the exception.

There are several advantages of policy gradient algorithms. According to Jan Peters' paper about policy gradient method[17], firstly the policy representations can be chosen to be meaningful. Secondly the parameters can incorporate previous domain knowledge. The third reason is that

policy gradient algorithm has a rather strong theoretical underpinning and additionally, policy gradient algorithms can be used in a model-free fashion.

All these advantages ensures that with only few parameters, robots can learn a decent policy for certain task. Mathematically, policy gradient algorithm tries to optimize policy parameters $\theta \in \mathbb{R}^n$ so that the expected return:

$$J(\theta) = \vec{E}\{\sum_{k=0}^{H} a_k r_k\} \tag{2.11}$$

is maximized. where a_k is a weighting factor. It can be set as γ^k in discounted case or $\frac{1}{H}$ in average case. The steepest decent algorithm is normally set to be optimization method as for each iteration, we would like to have small changes to the robot system.

$$\theta_{h+1} = \theta_h + \alpha_h \nabla_\theta J|_{\theta=\theta_h} \tag{2.12}$$

where α_h is learning rate for current updating step h.

2.3 Classification of the Regarded RL Problems

Many literatures have pointed out the complexity of RL problem in the area of robotics.[9][17]. Especially in Kober's paper[9], four different aspects were mentioned. He considers these four aspects as four curses for applying RL to robotics. These four curses are curse of High dimensionality, curse of real world samples, curse of under modelling and model uncertainty and curse of goal specification.

High-Dimensionality is the first characteristics considered in the area of robotics. Robot systems normally have many degrees of freedom(DOF), especially in modern anthropomorphic robots. For example, Baxter robot has two arms, each arm has 7 DOF including three pitch degrees and four roll degrees. This continuity makes traditional reinforcement learning fail as many traditional methods are based on discretization of each DOF. If the we discretize n DOFs and discretize each DOF to m states, the total states for the system is m^n, which is in inapplicable in most of the cases.

Need of **Real Samples** is another curse of applying RL to robotics. Robot system inherently interacts with physical system in real world. During test with environment, robot hardware may experience wear and tear. As a consequence, in many cases, this is an expansive process. Despite failure is costly, the test process also requires some kind of supervision from human. For example, we used Baxter to reach certain position as fast as possible. In optimization process, it stucks at a position that is hard to set.

If optimization process happens in more complex dynamic environment(e.g. helicopter robot), a supervision process conducted by several human's cooperation is needed. For all these reasons, generating real word samples require different resources and is a expansive process.

Under-modelling and model uncertainty is the next problem in robot system. For reducing the cost of real world samples, people build accurate simulators to accelerate learning process. Unfortunately, building such kind of models involves a lot of engineering work, which is also expansive. For small robot system, the simulator can improve learning process to some extend. But if we use simulator to simulate complex system, a small turbulence can cause learned system to diverge from real system.

Last but not the least, *goal specification* means to specify the reward function for the robot system. As in reinforcement learning algorithm, policy optimization process depends on the observing different rewards of two different policies. If a same reward is always received, there is no way of telling which policy is better. In practice, it is surprisingly difficult to specify the reward function of the system.

These four areas are notorious when people try to apply RL algorithm to robotics. Here we only discuss basic ideas of these problems. However, people who studies applying reinforcement learning in robotics have explored these problems more thoroughly than discussed here. If readers have interest, please refer to paper "Reinforcement learning in robotics: A survey"[9]

2.4 Policy Gradient with Parameter Exploration

As one of the policy gradient method, Policy Gradient with Parameter Exploration(PGPE) has shown its advantages in several scenarios. This algorithm was developed by Frank Sehnke and described in his paper "Policy Gradient with Parameter Exploration" [22]. PGPE algorithm follows basic idea of policy gradient algorithm, that is optimizing policy without using value estimation.

2.4.1 PGPE algorithm

As previously mentioned, the policy uses several policy parameters $\vec{\theta}$ to represent policy of the system. In the case of PGPE, the policy is represented as a simple linear model. For formalizing the algorithm, we will uses previous symbols as basis of system.

Considering a robot interacting with environment, at time t, the robot

is in a state $\vec{s_t}$, making action $\vec{a_t}$ and resulting in state $\vec{s}_{t+1}$ according a stochastic function $\vec{s}_{t+1} \sim p(\vec{s_{t+1}}|\vec{s_t}, \vec{a_t})$. Now with parameter $\vec{\theta}$, the action is defined as $\vec{a_t} \sim p(\vec{a}t|\vec{s}_t, \vec{\theta})$.

In PGPE, the action is determined by n weight matrices($W_i, \forall i \in \{1, 2, \cdots, n\}$). Mathematically, the action vector $\vec{a}$ is determined by :

$$\vec{a} = f(\vec{s}) \tag{2.13}$$

$$= \sum_{i}^{n} \vec{W}_i \cdot \vec{s} \tag{2.14}$$

where $f(\vec{s})$ is defined as $f(\vec{s})$.(sometimes, it is also called a multi-layer linear neural network.).

Each parameter in matrix W_{ijk} is defined by a Gaussian function:

$$W_{ijk} \sim \mathcal{N}(\mu, \sigma^2) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-\mu)^2/2\sigma^2} \tag{2.15}$$

where μ and $sigma$ are mean and variance of the algorithm, W_{ijk} means element of $j'th$ row and $k'th$ column of $i'th$ matrix.

The learning process of PGPE is a little bit different than other algorithm. Previously, the objective function is defined according to Formula 2.5. However, in episodic setting, the parameters are updated after each episode, which makes algorithm to depend on sampling of all parameters of this linear neuron network. After sampling parameters for all weights of network, we receive a sequence of all the reward r_t in this episode which results reward of this episode to be $r(h) = \sum_{t=1}^{T} r_t$. We then modify the algorithm to be:

$$J(\theta) = \int_H p(h|\vec{\theta}) r(h) dh \tag{2.16}$$

where H is set of all possible history h of this episode, h is defined as $h = [s_{1:T}, a_{1:T}]$.

Further expanding the formula by using standard identity $\nabla_x y(x) = y(x)\nabla_x \log(x)$:

$$\nabla_{\vec{\theta}} J(\vec{\theta}) = \int_H p(h|\vec{\theta}) \nabla_{\vec{\theta}} \log p(h|\vec{\theta}) r(h) dh \tag{2.17}$$

Since the whole process is Markovian process, i.e. we have $\vec{a_t} \sim p(\vec{a}_t|\vec{s}_t, \vec{\theta})$ defined as the transition function. The following property holds:

$$p(h|\vec{\theta}) = \prod_{i=1}^{T} p(\vec{s}_{i+1}|\vec{a}_i, \vec{s}_i) p(\vec{a}_i|\vec{s}_i, \vec{\theta}) p(\vec{s_0}) \tag{2.18}$$

$$\log(p(h|\vec{\theta})) = \sum_{i=1}^{T} \log(p(\vec{s}_{i+1}|\vec{a}_i, \vec{s}_i)p(\vec{a}_i|\vec{s}_i, \vec{\theta})) + \log(\vec{s_0}) \tag{2.19}$$

The term $\nabla_{\vec{\theta}} \log p(h|\vec{\theta})$ in the Formula 2.17 can be rewritten as:

$$\nabla_{\vec{\theta}} \log p(h|\vec{\theta}) = \nabla_{\vec{\theta}} \sum_{i=1}^{T} \log(p(\vec{s}_{i+1}|\vec{s}_i, \vec{\theta})) + \nabla_{\vec{\theta}} \log(\vec{s_0}) \tag{2.20}$$

$$= \nabla_{\vec{\theta}} \sum_{i=1}^{T} \log(p(\vec{s}_{i+1}|\vec{a}_i, \vec{s}_i)p(\vec{a}_i|\vec{s}_i, \vec{\theta})) \tag{2.21}$$

$$= \nabla_{\vec{\theta}} \sum_{i=1}^{T} \log(p(\vec{a}_i|\vec{s}_i, \vec{\theta})) \tag{2.22}$$

Then if we substitute the result of Formula 2.22 to Formula 2.17, we get more convenient form:

$$\nabla_{\vec{\theta}} J(\vec{\theta}) = \int_H p(h|\vec{\theta}) \nabla_{\vec{\theta}} \sum_{i=1}^{T} \log(p(\vec{a}_i|\vec{s}_i, \vec{\theta}))r(h)dh \tag{2.23}$$

In continuous case, it is impossible to get all the histories for certain set of policy parameters. We need to estimate the gradient based on the samples of this distribution as a result, we further modify our formulas to:

$$\nabla_{\vec{\theta}} J(\vec{\theta}) = \frac{1}{P} \sum_{p=1}^{P} \sum_{i=1}^{T} \nabla_{\vec{\theta}} \log(p(\vec{a_{ip}}|\vec{s_{ip}}, \vec{\theta_p}))r(h_p) \tag{2.24}$$

where, in formula, subscription p means $p'th$ sample of the distribution.

Previously, we discussed that, in PGPE, each parameter of matrices is determined by a Gaussian function as defined in Fomula 2.15. As we use weights matrices W_i as hidden policy parameters, the Formula 2.23 becomes as:

$$\nabla_{\vec{\mu},\vec{\sigma^2}} J(\vec{\theta}) = \int_H p(h|\vec{\theta}) \nabla_{\vec{\theta}} \sum_{i=1}^{T} \log(\int_\theta p(\vec{a}_i|\vec{s}_i, \vec{\theta}) p(\vec{\theta}|\vec{\mu}, \vec{\sigma^2}))r(h)dh \tag{2.25}$$

According to this formula, we can make a optimization process for both vectors of means and variances of Gaussian distribution. There are two ways of optimizing the function. One way is to calculate the gradient

information of each mean μ_i and gradient information of each σ with respect with respect to $\log(p(\theta|\vec{\mu}, \vec{\sigma^2}))$. From definition of Gaussian function we can get:

$$\begin{aligned}\nabla_\mu \log(p(\theta|\vec{\mu}, \vec{\sigma^2})) &= \nabla_\mu \log(\frac{1}{\sigma\sqrt{2\pi}} \exp \frac{-(\theta-\mu)}{2\sigma^2}) \\ &= \nabla_\mu - \frac{-(\mu-x)^2}{2\sigma^2} \\ &= \frac{\mu-x}{\sigma^2}\end{aligned} \tag{2.26}$$

and

$$\begin{aligned}\nabla_\sigma \log(p(\theta|\vec{\mu}, \vec{\sigma^2})) &= \nabla_\sigma \log(\frac{1}{\sigma\sqrt{2\pi}} \exp \frac{-(\theta-\mu)}{2\sigma^2}) \\ &= -\frac{1}{\sigma} + \frac{(x-\mu)^2}{\sigma^3} \\ &= \frac{(x-\mu)^2-\sigma^2}{\sigma^3}\end{aligned} \tag{2.27}$$

After getting gradient information of mean μ and variance σ^2 of system, we can use method proposed by Williams[25]. Instead of knowing exact step size, Williams uses a so-called reference reward to calculate the step size. Formula 2.28 2.29 shows the updating rules of mean and variances.

$$\Delta\mu = \alpha(r-b)\frac{\mu-x}{\sigma^2} \tag{2.28}$$

$$\Delta\sigma = \alpha(r-b)\frac{(x-\mu)^2-\sigma^2}{\sigma^3} \tag{2.29}$$

This method was influenced by Williams' REINFORCE algorithm as consequence, it inherits the advantages and disadvantages of this algorithm. On one hand, this method has proof of convergence, which means with more trials, this method will finally converge. On the anther hand, this method is also slow and unstable. A better solution based on Simultaneous Perturbation Stochastic Approximation(SPSA) provides fast convergence rate.

This method uses a symmetric sampling process for determining rewards of a specific mean μ and variance σ of random variable. Using SPSA, we first generate small perturbation ϵ from normal distribution $\mathcal{N}(0, \sigma^2)$. Then we then use two parameters $\vec{\theta}^+ = \vec{\mu} + \vec{\epsilon}$ and $\vec{\theta}^- = \vec{\mu} - \vec{\epsilon}$ for the system to get

two rewards r^+ and r^-. Combining with Formula 2.26 and Formula 2.27, we can get gradient information for $\vec{\mu}$ as:

$$\nabla_{\mu_i} J(\vec{\mu}, \vec{\sigma^2}) \approx \frac{\alpha\epsilon(r^+ + r^-)}{\sigma^2} \tag{2.30}$$

Then we use central different method and apply the same step size, then updating rule becomes:

$$\Delta_{u_i} = \alpha\epsilon\frac{(r^+ - r^-)}{2} \tag{2.31}$$

However, we can not use similar method for updating σ, since the selected two parameters θ^+ and θ^- have same variances. As a consequence, instead of using information from sample parameter ϵ along, we also consider the average value of two rewards to update σ. This updating algorithm is called SyS sampling method in some literature.[21]

$$\Delta_{\sigma_i} = \alpha\left(\frac{(r^+ + r^-)}{2} - b)\right)\left(\frac{\epsilon_i^2 - \sigma_i^2}{\sigma_i}\right) \tag{2.32}$$

where b is baseline and updated before updating hidden policy parameters $\vec{\mu}$ and $\vec{\sigma^2}$.

Formula 2.33 shows how baseline is updated. Some literature mentioned that variance of the base can affect quite a lot the learning speed of the system.[26]. But here we only use the simplest way of updating the algorithm.

$$b = 0.1 \cdot b + 0.9 \cdot \frac{r^+ + r^-}{2} \tag{2.33}$$

In the following section, we use pseudo code for whole PGPE algorithm.

Algorithm 2: Policy Gradient with Parameter Exploration

Data: The MDP Process Parameters
Result: Optimal Policy $\pi*$
while *average reward not converged* **do**
  Get mean $\vec{\mu}$ and variances $\vec{\sigma^2}$ of random variables.
  For each parameter, sample ϵ from Normal Distribution $\mathcal{N}(0, \sigma^2)$ and form a vector $\vec{\epsilon}$.
  Get two parameters using $\vec{\theta^-} = \vec{\mu} - \vec{\epsilon}$, $\vec{\theta^+} = \vec{\mu} + \vec{\epsilon}$
  Use two parameters as policy parameters, get cumulative reward r^+, r^- from system.
  Update baseline b of algorithm according to Formula 2.33.
  Update μ and σ using Formula 2.31 and Formula 2.32.

Chapter 3

Deep Recurrent Neural Networks

Recurrent Neural Network(RNN) is a special structure of neural network that has recurrent connections. And further more, deep recurrent neural networks are a neural network models that have more than one hidden layer. In the following sections of this chapter, we are going to discuss the details of this kind of model.

3.1 Deep Learning and its Recent Advances

Deep learning attracted a lot of attention since 2006. In year 2006, Geoffrey Hinton, one of the founders of idea deep learning, published a paper called "Reducing the Dimensionality of Data with Neural Networks"[6]. In this paper, Hinton and Slakhutdinow showed how a many-layered feedforward neural network can be pre-trained layer by layer. Since this paper, the work "deep learning" become famous in the community. However, before Hinton's work, several deep learning algorithms were developed. One of the initial works belongs to Kunihiko Fukushima who invented a model called Neocognitron in 1980. After that, in 1989, Yann Lecun was able to use an optimization algorithm called back-propagation to train the neuron network. Then Yann Lecun further simplified network to be so-called *Convolutional Neuron Network(CNN)* in 1998. As CNN becomes so successful in the area of computer vision, the deep learning embraced boost after that period. However, other models that has more than one hidden layer is also normally discussed in the context of deep learning. For example, Deep Belief Network (DBN) and Deep Boltzmann Machine(DBM) are not normally considered as neuron network models, but they also have application in the area of computer vision.

3.2 Feedforward Neural Networks

In year 1957, the simplest structure of neural network called perceptron was introduced by Frank Rosenblatt in Conell's Aeronautical Laboratory. This model consists only one cell and multiple connections. Figure **??** shows the basic structure of perceptron.

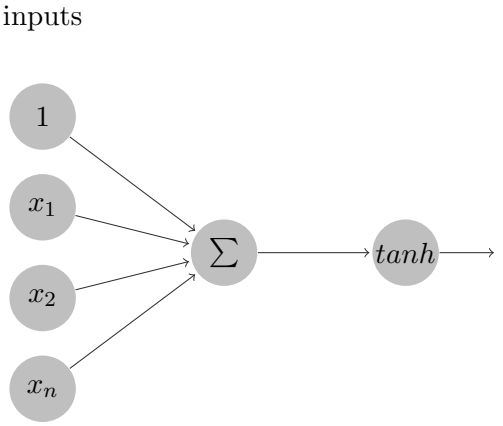

Figure 3.1: This picture shows the structure of perceptron, the i'th input is shown as x_i and corresponded weights are denoted as w_i. There is also a bias term name b connected to cell, which, with a addition function, generates the output.

The formula of representing input and output is defined as:

$$y = \vec{w} \cdot \vec{x} + b \tag{3.1}$$

If we consider b also as one of the inputs, then the formula can be defined as:

$$y = \vec{w}_{new} \cdot \vec{x}_{new} \tag{3.2}$$

where $\vec{w}_{new} = [\vec{w}; b]$ and $\vec{x} = [\vec{x}; 1]$.

This simple structure was considered promising initially from several points of view. But after further investigation, the perceptron was proven that it can not classify many non-linear classes of patterns. However, its discovery leads to a field of research called neural networks in area artificial intelligence. As a consequence, since 1957, people started trying different methods for modifying this model to adapt to different problems. One

important modification is to add a non-linear transformation function to the system i.e. after getting y from Formula**??**, we use a function like tanh to get a new $\hat{y}$ to ensure the output is restricted in a range. Anther important modification of the system is to stack many perceptrons together to build a large and complex model for the classification purpose. This kind of networks is normally called Feed-forward Neural Networks(FFNN) as information send to this system is propagated only from lower layer to higher layer. Figure 3.3 shows the structure of FFNN system.

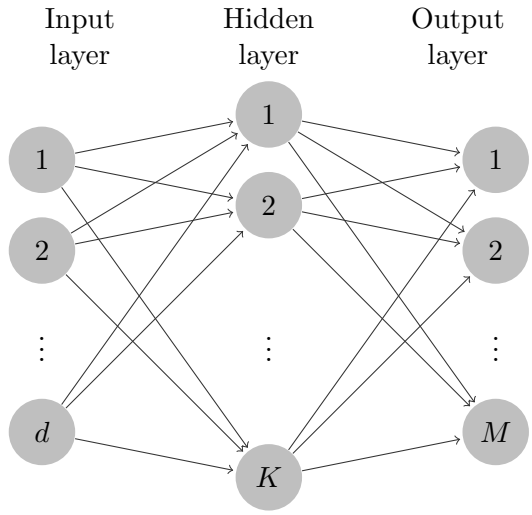

Figure 3.2: This picture shows the structure of Feed-forward Neural Network. There are three layers in this network, namely the input layer, hidden layer and the output layer. For connections from input layer to hidden layer, the i'th input is shown as x_i and corresponded weights of hidden neuron j is denoted as W_{ij}.

The formula describing the each layer is then defined as:

$$f(\vec{x}) = \sigma(W\vec{x}) \tag{3.3}$$

where $\sigma(\cdot)$ is a nonlinear function.(e.g. tanh)

3.3 Recurrent Neural Networks

The Recurrent Neural Networks(RNN) contains at least one neuron that has at least one recurrent connection i.e. a connection that connects to itself or to lower layer. This special structure makes memory cell to be internal

memory which enables network to memories the change of sequential data. Unlike the feed-forward neural network, the recurrent neural network is used for predicting next data point.

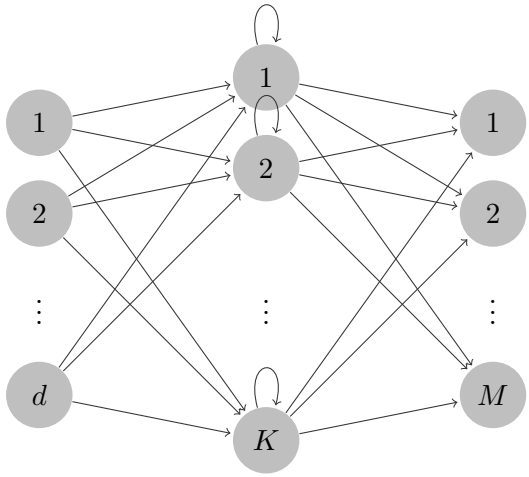

Figure 3.3: This picture shows the structure of recurrent neural network. There are two layers in this networ namely input layer and hidden layer, the first layer contains notes connected to the input data. The second layer stores information and also forwards information to next layer. The cell in the hidden layer has a connection to itself which means the information stored in the cell at time $t-1$ also influences the information stored in the cell at time t

3.3.1 Finite Unfolding in Time

When considering the structure of neural network, people normally application Finite Unfolding in Time for RNN. It means to introduce another dimension for RNN. In this way, the recurrent connection can be dealt with more easily. In the following section, we will use a simple example for explaining this idea. The model we are going to use contains one input neuron, one hidden neuron and one output neuron. See Figure 3.4 for reference.

In this figure, we mark input layer as $\vec{x}$, hidden layer as $\vec{s}$ and output layer as $\vec{o}$.

The basics idea of unfolding RNN in time is to copy RNN several times and connect them in a chronological order. If the connection is recurrent, then the connection should be connect to the same neuron of next time

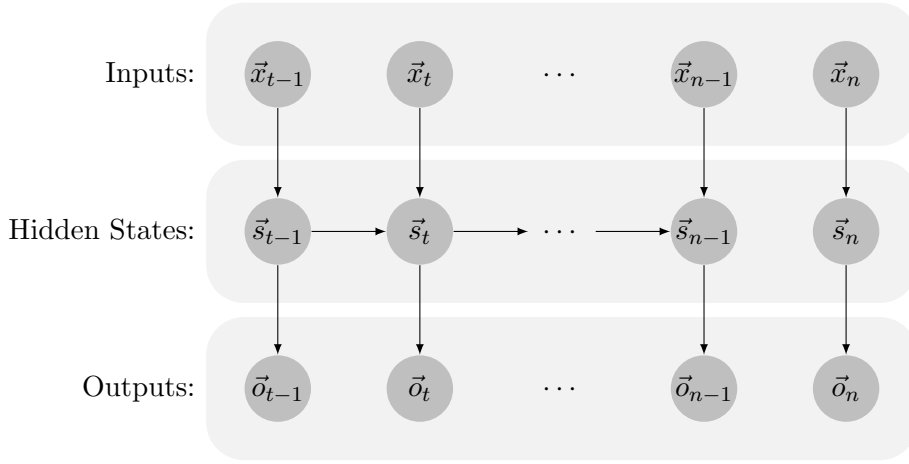

Figure 3.4: This picture shows the structure of a simple RNN. It contains three layers i.e. one input layer, one hidden layer and one output layer.

step.Figure 3.5 illustrates the model of unfolding simple RNN for three time steps.

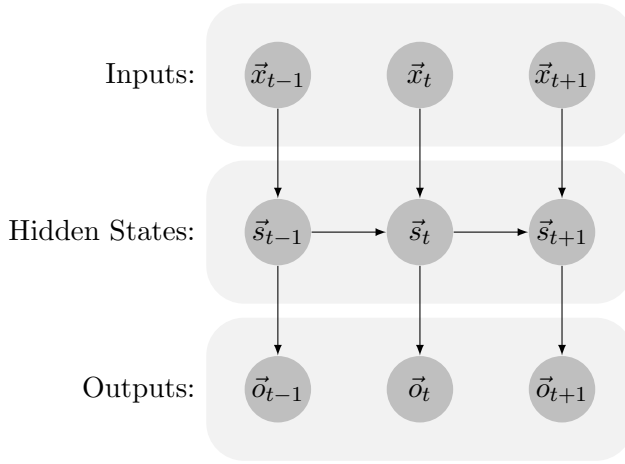

Figure 3.5: This picture contains model that unfolds a simple RNN described in Figure3.4 in abitary n time steps. In this figure, x_t represents input layer at time t, s_t representes the hiddent states at time t and o_t representes output layer at time t.

The finite unfolding technique transforms a neural network with recurrent connection to a network that is easier to compute gradients. It is also

easy for us to write this neural network's expression.

$$\vec{s_t} = \sigma(W\vec{x}_t + B\vec{s}_{t-1}) \tag{3.4}$$

Where $\vec{s_t}$ is the value of hidden state at time t and B is the weight matrix for updating hidden state information from time $t-1$ to t.

3.3.2 Overshooting

Considering that we only one time series, the task for RNN is to predict next data point based on previous data we have. There are two steps for solving this problem. First we need to copy the data into two set and shift the input by one to produce output. It is illustrated as shown in Figure 3.6.

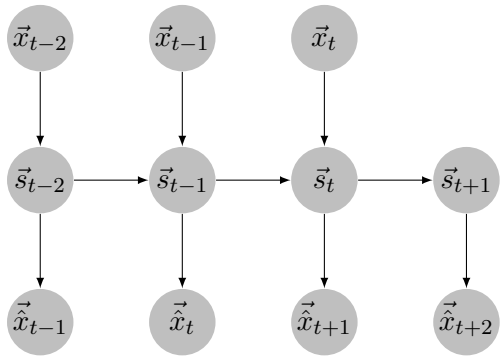

Figure 3.6: This figure isllustrates in a predictive model made by RNN, how does it generate $\hat{x_t}$ based on previous data. After using all training example, the model lacks input data. Then the neural caluclation of hidden defined as $\vec{s_t} = \sigma(W\vec{x_t} + B\vec{s_{t-1}}))$ becomes $\vec{s_t} = \sigma(B\vec{s_{t-1}})$

Last training step takes input $\vec{x}(t-2)$, $\vec{x}(t-1)$, $\vec{x}(t)$ as input and produce $\vec{x}(t-1)$, $\vec{x}(t)$, $\vec{x}(t+1)$ as outputs. However it is very difficult to predict value at time t+2 as we don't have information of $\vec{x}(t+1)$. According to the Fomula 3.4, the input is set to $\vec{0} \quad \forall t > T$ then we get:

$$\vec{s_t} = \sigma(B\vec{s_{t-1}}) \tag{3.5}$$

As a consequence, the output defined as $\vec{x_{t+1}} = A\vec{s_t}$ becomes the prediction for next value. This phenomenon is called overshooting.

3.3.3 Dynamical Consistency

Dynamical consistency is kept by introducing the output of last time step to input of current time step. Figure 3.8 illustrates how is it done through the modification of structure.

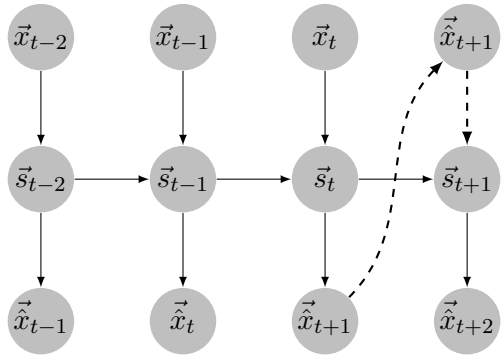

Figure 3.7: This picture contains model for predicting data for time series. For the first unfolded structure of neural network, the data of series at time t-2 is fed to the network and we expect data at time t-1 are predict by the network.

By remembering all the parameters in the network and unfold in time for several steps, the recurrent neural network is able to predict the next data point in the sequence. The relationship between input and output is listed as follows:

$$\vec{s_t} = \sigma(W\vec{x_t} + B\vec{s_{t-1}}) \tag{3.6}$$
$$\vec{x_{t+1}} = A\vec{s_t} \tag{3.7}$$

where A means weight matrix of internal states to output states.
To sum up, the dynamics of system becomes as follows.

$$\vec{s_t} = \sigma(W\vec{x_t} + B\vec{s}_{t-1}) \quad \forall t \leq T \tag{3.8}$$
$$\vec{s_t} = \sigma(W\vec{\hat{x}}_t + B\vec{s}_{t-1}) \quad \forall t > T \tag{3.9}$$
$$\vec{o_{t+1}} = A\vec{s_t} \quad \forall t \leq T-1 \tag{3.10}$$
$$\vec{\hat{x}}_{t+1} = A\vec{s_t} \quad \forall t > T-1 \tag{3.11}$$

Formula 3.5 shows how are values of internal states updated at the training stage. Formula 3.9 shows how are values of output data predicted after

training stage. Formula 3.10 shows how outputs generated after training stage. Formula 3.11 shows how outputs predicted after training stage. The overall goal of the system is to minimize the different between the output and predicted output for all time series. Mathematically, it is defined as:

$$J = \sum_{i=1}^{T} \vec{o}_t - \vec{y}_t \tag{3.12}$$

If we assume each input and output has N data points, then the cost becomes:

$$J = \sum_{i=1}^{T} \sum_{n=1}^{N} o_t^n - y_t^n \tag{3.13}$$

The main goal of function is to minimize the cost over function parameters defined in the network.(See section 3.5 for more reference.)

3.4 Universal Approximation

It has been proven that multi-layer feed-forward neural networks are universal approximators by Hornik in 1989. Similar works have also been mentioned by Cybenko and Funahashi in the same year. In this work, Hornik continued work of Minskey and Papert about two layers network. Minkey and Papert proved that two-layer neural networks are not able to approximate functions that do not belong to a special class. Then by adding a third layer, the neural network is able to approximate different functions.

Details of approximation theory of multi-layer neuron network is governed by Stone-Weierstrass theorem. This theory states that every continuous function defined on a closed interval $[a, b]$ can be uniformly approximated by a polynomial function. As a result, it also proves that a neuron network with more than one hidden layer can approximate any function.

RNN, as special model of neuron network model also follows Stone-Weierstrass theorem. Herrn Anton Maximilian Schafer proved that RNN is also a universal approximator.

3.5 Training of RNN

After getting data and designing the structure of network, the next important step is to train the network to model the data we have and also to predict next point in the data sequence. The main goal of training is to

minimize the cost of objective function between expected outcome and real outcome. As a result, predicting data based on data we have.

3.5.1 Shared Weight Extended Backpropagation

Training algorithm of recurrent neural network, called back-propagation, is introduced from feed-forward neural network. For simplicity of this algorithm, we first consider a simple feed-forward neural network defined as:

$$y = \sigma(W\vec{x}) \tag{3.14}$$

where W is a 3×4 matrix, $\vec{x}$ is a vector of length 4 and $\sigma(\cdot)$ is an element-wise non-linear transformation.

This network only contains two layers including input layer and output layer. Input layer has four input neurons and output layer has three output neurons. The data we have is a list of input-output pairs.

Formally the backpropagation algorithm is defined as:

Algorithm 3: Backpropagation for two layers feed-forward neural network.

Data: List of input-output pairs
Result: return the network with parameters W
initialize network weights (often small random values);
while *training example ex* **do**
    prediction = neural-net-output(network, ex);
    actual = teacher-output(ex);
    compute error (prediction - actual) at the output units;
    compute ΔW for all weights from hidden layer to input layer;
    update network weights ;
    until all examples classified correctly or another stopping criterion satisfied

The key element in this algorithm is the updating rule of the system. Based on

3.5.2 Learning Long-Term Dependencies

Long-term dependences is important in control. Something happened at one particular time might has an influences at another time. If one system can identify these kinds of moment, it would be beneficial to to predict outcome influenced by environment. Although there are other algorithm

like policy gradients and Optimal Ordered Problem Solver also can do this job for control, some specially designed RNN can remember long-term dependences under a flexible setting.

Transitional RNN has several problems. One famous problem is that it suffers exploding gradient and vanishing gradient problem as when RNN unfolds in time, it becomes neurally deep. The gradient becomes very small as we back-propagate it to first few layers.

LSTM is normally treated as a hidden layer in a neural network system. It has input connections, three gate, outputs connection. A more illustrative example is shown in the following figure.

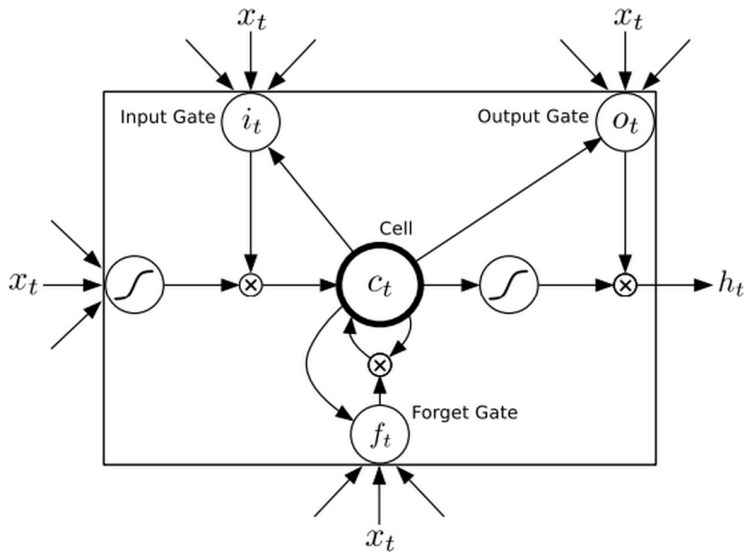

Figure 3.8: This figure shows the basic strucuture of LSTM. In the middel, there is the memory cell which keeps the information of the data sequence. Around it, there are three gates, namely input gate, output gate and foget gate. Each of them gets information from the input and controls the updatting rule of memory cell. At the left side, there is input connection. On right side, there is output connection of this layer.

The idea of LSTM is relatively straight forward. The memory cell stores information of the sequential data and gates in this layer tries to when and how much this information should be updated.

The basic gradient descent approach (and its backpropagation algorithm implementation) is notorious for slow convergence, because the learning rate γ must be typically chosen small to avoid instability. Many speed-

up techniques are described in the literature,

- dynamic learning rate adaptation schemes. (complexity $O(TM)$, where T is number of epoch and M is number of connections)
- use second-order gradient descent techniques, which exploit curvature of the gradient but have epoch complexity $O(TM^2)$.
- local error minimum
- adding noise
- by repeating the entire learning from different initial weight settings
- using task-specific prior information to start from an already plausible set of weights

Starting from LSTM section
* more complex unit is called a *memory cell* * there are many memory cells. We mark j'th cell as c_j. * c_j gets input from net_{c_j}, out_j(output gate), in_j(input gate). * in_j's activation at time t is denoted as $y^{in_j}(t)$ and we denote out_j's activation at time t is denoted as $y^{out_j}(t)$

$$g^{out_j}(t) = f_{out_j}(net_{out_j}(t)) \tag{3.15}$$

where $f_{out_j}(\cdot(t))$ means activation function of out gate of memory cell c_j.

similarly:

$$g^{in_j}(t) = f_{in_j}(net_{int_j}(t)) \tag{3.16}$$

where $f_{in_j}(\cdot(t))$ means activation function of out gate of memory cell c_j.

and

$$net_{out_j}(t) = \sum_u w_{out_j u} y^u(t-1) \tag{3.17}$$

and

$$net_{in_j}(t) = \sum_u w_{in_j u} y^u(t-1) \tag{3.18}$$

We also have

$$net_{c_j}(t) = \sum_u w_{c_j u} y^u(t-1) \tag{3.19}$$

In above *Equation* 3.17, *equation* 3.18, *equation* 3.19, w_j means the weight of connection from j'th node to c_j

As a consequence, the $w_{out_j u}$ means for the outgate of $j'th$ memory cell, the wight of $u'th$ connection conneted to memory cell c_j and it is similar to all of the connections.

* All these dierent types of units may convey useful information about the current state of the net. * an input gate (output gate) may use inputs from other memory cells to decide whether to store(access) certain information in its memory cell. * There even may be recurrent self-connections like $w_{c_j c_j}$.

We have considered three conponents of memory cell. Now we consider the output of the memory cell. The output of c_j is defined as:

$$y^{c_j}(t) = y^{out_j}(t) h(s_{c_j}(t)) \tag{3.20}$$

Where $y^{c_j}(t)$ is the output of memory cell c_j at time t. $y^{out_j}(t)$ is the output of outgate at time t. h is a sigmoid function that scales the state of self-connected node which is center component of cell memory(we also call it as inner state of memory cell).

The inner state of memory cell c_j is defined as:

$$s_{c_j}(t) = s_{c_j}(t-1) + g(t-1) \cdot y^{in_j}(t-1) \tag{3.21}$$

Where $g(t-1) = g(net_{cj}(t))$, $g(\cdot)$ is the activation function of new input.

* memory cell c_j (the box) and its gate units in_j, out_j builds *constant error carrousel*.(CEC) * Distributed output representations typically do require output gates. Not always are both gate types necessary, though one may be sucient. For instance, in Experiments 2a and 2b inSection 5, it will be possible to use input gates only. * outputgates can be benecial: they prevent the net's attempts at storing long time lag memories

Following is the picture of the structure of the system.

3.5.3 Optimization Methods

Optimizing neuron network is not a simple task. People tried different method since perceptron was introduced. However, only until the back-propagation algorithm was introduced, the neuron network models have a role in the machine learning field. Currently, back-propagation has become standard of training neuron network and many optimization algorithms for neuron network are based on that. In the following section, I will describe two optimization methods I used in the experiments. One of them is call *Nesterov's Accelerated Gradient*(NAG) and another of them is called *Adam*.

Nesterovs Accelerated Gradient

Nesterov's Accelerated Gradient is sometimes called Nesterov's momentum. Here momentum refers to Classic Momentum(CM) method which is a technique to accelerate the standard gradient descent algorithm. Let recall the original gradient descent algorithm, which update parameters based on a step size α and gradient information of objective function. Formula 3.22

$$\Delta_\theta = -\alpha \nabla_{\vec{\theta}} f(\vec{\theta}) \tag{3.22}$$

CM method adds a momentum coefficient μ to system and use this parameter to include a momentum-like behaviour in the optimization process. The updating rule for parameters become influenced by the momentum as shown in Formula 3.23.

$$\widetilde{\Delta}_{\theta t} = \mu \widetilde{\Delta}_{\theta t-1} - \alpha \nabla_{\vec{\theta}} f(\vec{\theta}) \tag{3.23}$$

NAG, like CM algorithm, is a first order optimization algorithm. For a general smooth(non-strongly) convex functions and deterministic gradient, NAG has a convergence rate of $O(\frac{1}{T^2})$, where T means optimization steps. Comparing convergence rate of $O(\frac{1}{T})$ for CM, it is exponentially faster. The updating rule for NAG is show in Formula 3.24 .In fact, there is not much difference between CM and NAG.

$$\widetilde{\Delta}_{\theta t} = \mu \widetilde{\Delta}_{\theta t-1} - \alpha \nabla_{\vec{\theta}} f(\vec{\theta} + \mu \widetilde{\Delta}_{\theta t-1}) \tag{3.24}$$

Adam

Adam is a first-order gradient-based algorithm [8] invented by Diederik Kingma and Jimmy Ba in 2015. It combines ideas of RMSProp and AdaGrad. Algorithm 4 shows the each step of this algorithm.

Algorithm 4: *Adam*, g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$ and $\lambda = 1 - 10^{-8}$.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$, $\lambda \in [0, 1]$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
$m_0 \leftarrow 0$ (Initialize initial 1^{st} moment vector)
$v_0 \leftarrow 0$ (Initialize initial 2^{nd} moment vector)
$t \leftarrow 0$ (Initialize timestep)
while θ_t not converged **do**
$t \leftarrow t + 1$
$\beta_{1,t} \leftarrow \beta_1 \lambda^{t-1}$ (Decay the first moment running average coefficient)
$g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
$m_t \leftarrow \beta_{1,t} \cdot m_{t-1} + (1 - \beta_{1,t}) \cdot g_t$ (Update biased first moment estimate)
$v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
$\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
$\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
$\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
end while
return θ_t (Resulting parameters)

Chapter 4

Prior Arts of Combining Deep Neuron Network and RL

Different research groups have tried different methods for combining deep neuron networks with reinforcement learning. The first thoughts turned out to be using neuron network as a general controller. In the early work applying neuron network to control algorithm, several algorithms using neuron network were proposed including Explanation-Based Neural Network Learning[23] and Neural networks for self-learning control systems. [15]. These early initiatives provide information about how could one formulate a

4.1 Deep Q Network

Deep Q Network is a network proposed by Google's deep learning group. It creates a direction of combining Convolutional Neuron Network with Q learning algorithm[13][14].

Chapter 5

Experiment

5.1 Cart-pole Balancing

Cart-pole balancing problem has been a basic test problem in the control theory. It considers a physical agent that tries to balance a pole attached to it. Figure 5.1 illustrates the situation of the system. The system consist several states including the position s and velocity $\dot{s}$ of the cart, angle θ and angular velocity $\dot{\theta}$ of the pole. The task involved in this process is to apply a suitable force to enable the system to balance the pole L so that the angle θ and position of cart s are kept in a range.

Based on the information we have about the system, we can write the Lagrangian of system.

$$L = \frac{1}{2}M\dot{s}^2 + \frac{1}{2}mv_m^2 - mgl\cos(\theta) \tag{5.1}$$

where, in this case, the v_m is the speed of the a mass attached to pole L. We infer from system

$$v_m^2 = \left(\frac{d}{dt}(x - l\sin(\theta)\right)^2 + \left(\frac{d}{dt}(l\cos(\theta)\right)^2 \tag{5.2}$$

Then it can be further simplified as

$$v_m^2 = \dot{s}^2 - 2l\dot{x}\dot{\theta}\cos(\theta) + l^2\dot{\theta}^2 \tag{5.3}$$

Following the definition of Lagrange the equations of motion are defined as:

$$\frac{d}{ds}\frac{\partial L}{\partial \dot{s}} - \frac{\partial L}{\partial s} = F \tag{5.4}$$

$$\frac{d}{dt}\frac{\partial L}{\partial \dot{\theta}} - \frac{\partial L}{\partial \theta} = 0 \tag{5.5}$$

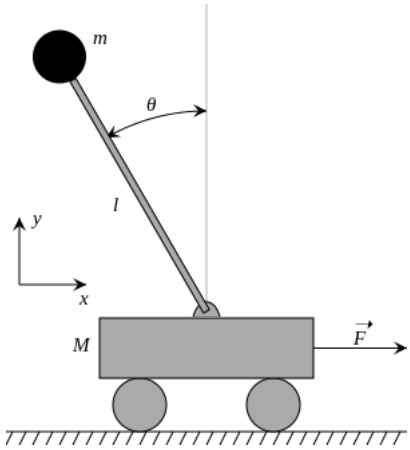

Figure 5.1: This figure shows the cart-pole balancing system. A force $\vec{F}$ is exerted on the cart C with mass M to balance a pole l that also has length l attached to it. A mass m is attached on pole L. The system has four states including point of the system s, velocity of the system $\dot{s}$, angel between pole L and normal θ and angular acceleration $\dot{\theta}$. When force $\vec{F}$ is exerted on cart C, the L will also have a force on the point of attachment with cart and the task is to provide forces that can keep position of cart as well as angel θ between pole and normal in a range of allowed values(e.g. $s \in [-4, 4]$, $\theta \in [-0.2, 0.2]$)

If we substitute the Lagrangian into system, we find the two equations of motion become like

$$(M+m)\ddot{s} - ml\ddot{\theta}\cos(\theta) + \frac{1}{2}ml^2\dot{\theta}^2 - mgl\cos(\theta) = F \tag{5.6}$$

$$l\ddot{\theta} - g\sin(\theta) = \ddot{s}\cos(\theta) \tag{5.7}$$

Traditional control algorithms like proportional-integral-derivative controller (PID) control can solve problem with high frequency loop. Although it is an entrance level problem in control theory and also in reinforcement learning, it is also a basic example to test effectiveness of the algorithm.

We used this model for testing the basic performance of the policy gradient algorithm. In this case, the policy algorithm also needs to take into account the states and evaluate the action. During the process, it needs to update it policy parameters from experience gained from its failures.

Normally the process' reward is calculated by simulator during the training of actor network. However, at mean time, we also learn a model that can also predict the reward and action according to the previous we

gathered.

5.1.1 System Implementation

The system is implemented using a several libraries of python including Pybrain, Theano and nntools to provide easy use of the testing process.

Pybrain is library developed by IDSIA [20] in order to offer flexible, easy-to-use yet still powerful algorithms for machine learning tasks. It is excellent for reinforcement learning researchers as it separates reinforcement learning into three parts namely, environment, agent and task. In this way, the library actually builds a framework for further development.

Theano is a GPU-based computational library for implementing deep learning algorithm. It has complete support for different deep learning structures including Convolutional Neuron Networks(CNN), Deep Blief Network(DBN) or Restricted Boltzmann Machine(RBM). It builds itself on scipy, which is a scientific computing library of python and numpy, which is a numerical computation library of python. This enables the library to provide the best computation performance for the system. It also uses Cuda-toolkit to support GPU-based computation, which ensures the nvidia-based system can have best performance.

nntools is library built on Theano. It provides modularized APIs for different deep learning architectures.

We use these libraries to build reinforcement learning platform. Following the the MDP process, we need to figure out four important elements of MDP process, namely, $(\mathbb{S},\mathbb{A},T\cdot(\cdot,\cdot),R\cdot(\cdot,\cdot))$. $\mathbb{S}$ is the set of all states of the system. Each state $s \in \mathbb{S}$ contains four physical parameters including s position of the cart, $\dot{s}$ velocity of the cart, θ angle of pole, $\dot{\theta}$ angular velocity of the pole. The action $a \in \mathbb{A}$ is decided by policy parameters and current states. Transition probability T is decided by the physical system and reward function R is defined as following:

$$R(s,\theta,t) = \begin{cases} 0, & \text{if} |s| \leq 0.05, |\theta| \leq 0.05, t \leq 200 \\ -1, & 2.4 \geq |s| \geq 0.05, 0.7 \geq |\theta| \geq 0.05, t \leq 200 \\ -2 \cdot (200 - t), & \text{otherwise} \end{cases} \tag{5.8}$$

Where t is the time steps that system has been running.

5.1.2 Experiment Result

The result of the algorithm for solving a cart-pole balancing problem is great. Within 300 trials, it can find best policy parameters for the system.

However, it needs a little bit longer to get best average rewards. Figure 5.2 shows the performance of the PGPE algorithm on cart-pole balancing problem.

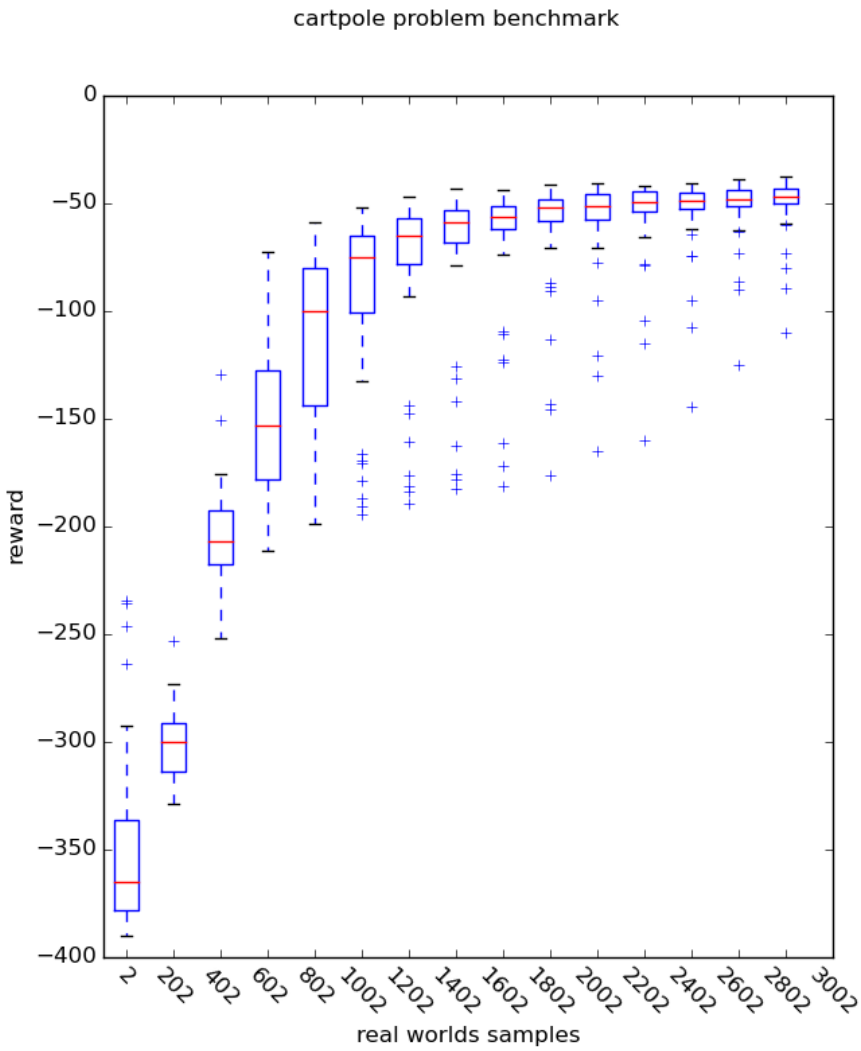

Figure 5.2: This figure shows the result of benchmark of cart-pole balancing problem. The x-axis represents the real word samples needed for training and the y-axis represents the average reward of 50 trials.

5.2 Baxter Robot Learning a Action

Baxter robot is a research robot developed by rethink company(as shown in Figure 5.2).It has two arms, two gripper and animated face. In this experiment, we will use its arms and gripper to learn to perform a certain wood stacking task. In following sections, we will discuss how is this experiment implemented.

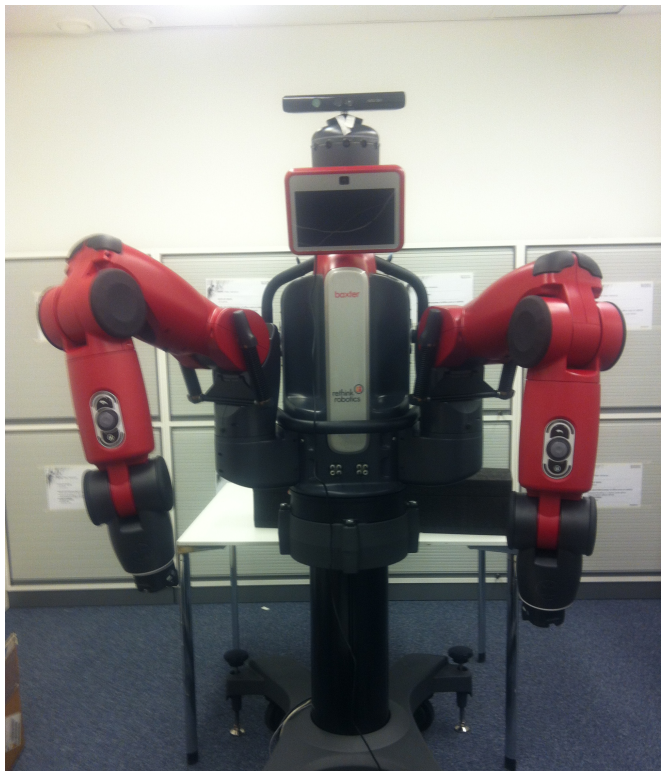

Figure 5.3: This figure shows a baxter research robot. We can see that it has a face and two arms.

5.2.1 System Implementation

Each arm of Baxter contains seven degrees of freedom including four roll degrees of freedom s_0, e_0, w_0 and w_2, three pitch degrees of freedom s_1 , e_1 w_1. The detail of the information is presented in picture. In total, Baxter's arms have fourteen degrees of freedom, which we consider as states of robot. They together form one state of the robot i.e. $(\{s_0^l, s_1^l, e_0^l, e_1^l, w_0^l, w_1^l, w_2^l\} \cup$

$\{s_0^r, s_1^r, e_0^r, e_1^r, w_0^r, w_1^r, w_2^r\}) \in \mathbb{S}$. Then we can apply each degree of freedom an angular velocity as action, as a result, we have a set of actions $a \in \mathbb{A}$. The policy function $\pi(\cdot)$ is a n-layer multilayer linear neuron network that mathematically formulated as:

$$\pi(\vec{s}) = (f_1 \circ f_2 \cdots f_n)(\vec{s}) \tag{5.9}$$

where $f_i(\vec{s}), \forall i \in \{1 \cdots n\}$ is defined as $W_i \cdot \vec{s}$ as a single layer linear neuron network. The transition function follows physical law with some noise. The reward function is defined similar as cart-pole problem. Now we follow the formulation of PGPE algorithm, using two hidden parameters μ, σ^2 for the each parameters in policy function. We also update algorithm according to PGPE algorithm **??** mentioned in section Reinforcement Learning. Similarly, we also define the reward function as a step function. Only here we need to define time steps to be longer, which can be seen as Formula

$$R(s, \theta, t) = \begin{cases} 0, & \text{if} |s| \leq 0.05, |\theta| \leq 0.05, t \leq 2000 \\ -1, & 2.4 \geq |s| \geq 0.05, 0.7 \geq |\theta| \geq 0.05, t \leq 2000 \\ -2 \cdot (2000 - t), & \text{otherwise} \end{cases} \tag{5.10}$$

5.2.2 Experiment Result Using Baxter Simulator

Based on the Robot Operating System(ROS), several simulators supports simulation of Bater robot. One of them is Gazebo, an official simulator provided by ROS.

5.2.3 Experiment Result Using Baxter Robot

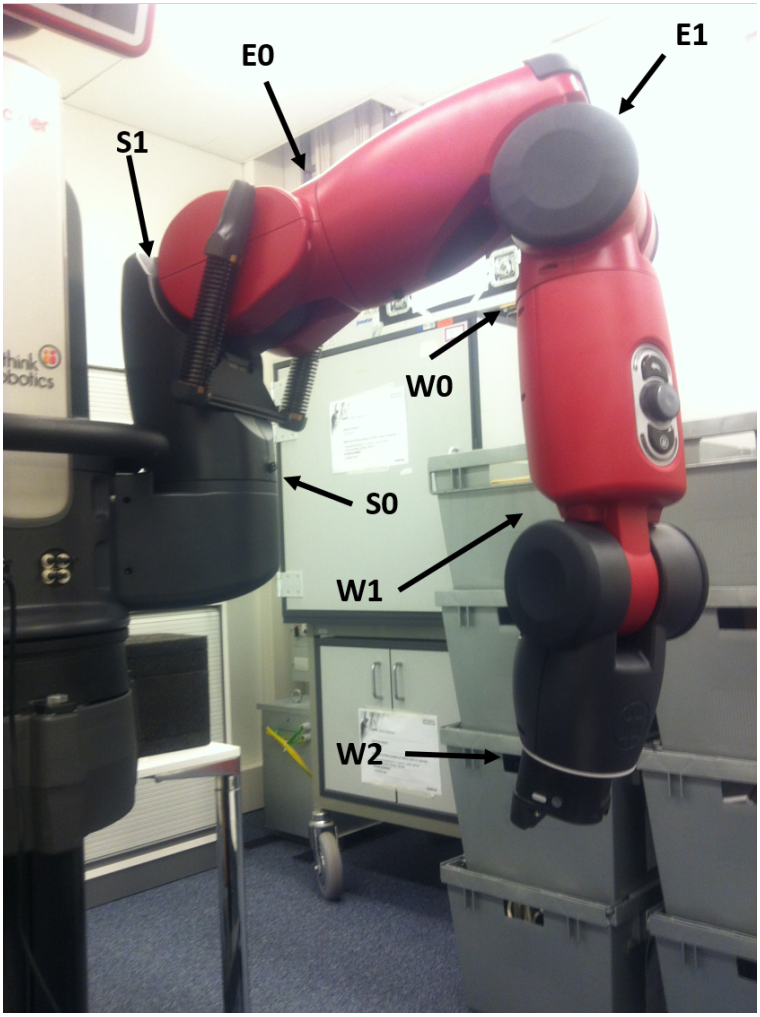

Figure 5.4: This picture shows the Baxter robot and its corresponding joints. Seven degrees of freedom including shoulder roll s_0, elbow roll e_0, wrist roll w_0, wrist roll w_2, shoulder pitch s_1 , elbow pitch e_1 and wrist pitch w_1 are presented on picture.

Chapter 6

Future Research

6.1 Combining Control with Vision Using Deep Neural Network

- One may use unified neural network model for vision and control together.
- Deep neural network archived state-of-art or close to state-of-art result in video recognition tasks.(also in)

6.2 Transfer Learning for Learning Repetitive Behaviour Using Deep Neural Network

References

[1] K. Cho, B. van Merrienboer, C. Gulcehre, F. Bougares, H. Schwenk, and Y. Bengio, *Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation*, arXiv, (2014).

[2] A. Graves, *Generating sequences with recurrent neural networks*, arXiv preprint arXiv:1308.0850, (2013), pp. 1–43.

[3] A. Graves, G. Wayne, and I. Danihelka, *Neural Turing Machines*, (2014), pp. 1–26.

[4] E. Guizzo, *How google's self-driving car works*, IEEE Spectrum Online, October, 18 (2011).

[5] G. E. Hinton, S. Osindero, and Y. W. Teh, *A fast learning algorithm for deep belief nets.*, Neural computation, 18 (2006), pp. 1527–54.

[6] G. E. Hinton and R. R. Salakhutdinov, *Reducing the dimensionality of data with neural networks*, Science, 313 (2006), pp. 504–507.

[7] J. Hoffman, S. Guadarrama, E. Tzeng, R. Hu, J. Donahue, R. Girshick, T. Darrell, and K. Saenko, *LSDA: Large Scale Detection Through Adaptation*, (2014), pp. 1–9.

[8] D. Kingma and J. Ba, *Adam: A method for stochastic optimization*, arXiv preprint arXiv:1412.6980, (2014).

[9] J. Kober, J. a. Bagnell, and J. Peters, *Reinforcement learning in robotics: A survey*, The International Journal of Robotics Research, 32 (2013), pp. 1238–1274.

[10] A. Krizhevsky, I. Sutskever, and G. E. Hinton, *ImageNet Classification with Deep Convolutional Neural Networks*, Advances In Neural Information Processing Systems, (2012), pp. 1–9.

[11] I. Lenz, H. Lee, and A. Saxena, *Deep Learning for Detecting Robotic Grasps*, CoRR, abs/1301.3 (2013).

[12] H. Mayer, F. Gomez, D. Wierstra, I. Nagy, A. Knoll, and J. Schmidhuber, *A system for robotic heart surgery that learns to tie knots using recurrent neural networks*, in IEEE International Conference on Intelligent Robots and Systems, 2006, pp. 543–548.

[13] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, *Playing atari with deep reinforcement learning*, arXiv preprint arXiv:1312.5602, (2013).

[14] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al., *Human-level control through deep reinforcement learning*, Nature, 518 (2015), pp. 529–533.

[15] D. H. Nguyen and B. Widrow, *Neural networks for self-learning control systems*, Control Systems Magazine, IEEE, 10 (1990), pp. 18–23.

[16] P. O'Connor, D. Neil, S. C. Liu, T. Delbruck, and M. Pfeiffer, *Real-time classification and sensor fusion with a spiking deep belief network*, Frontiers in Neuroscience, (2013).

[17] J. Peters and S. Schaal, *Policy gradient methods for robotics*, in Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on, IEEE, 2006, pp. 2219–2225.

[18] M. Raibert, K. Blankespoor, G. Nelson, and R. Playter, *BigDog , the Rough-Terrain Quaduped Robot*, tech. rep., 2008.

[19] R. Salakhutdinov and G. Hinton, *Deep Boltzmann Machines*, Artificial Intelligence, 5 (2009), pp. 448–455.

[20] T. Schaul, J. Bayer, D. Wierstra, Y. Sun, M. Felder, F. Sehnke, T. Rückstiess, and J. Schmidhuber, *PyBrain*, Journal of Machine Learning Research, 11 (2010), pp. 743–746.

[21] F. Sehnke, *Efficient baseline-free sampling in parameter exploring policy gradients: Super symmetric pgpe*, in Artificial Neural Networks and Machine Learning–ICANN 2013, Springer, 2013, pp. 130–137.

[22] F. Sehnke, C. Osendorfer, T. Rückstiess, A. Graves, J. Peters, and J. Schmidhuber, *Policy gradients with parameter-based exploration for control*, in Artificial Neural Networks-ICANN 2008, Springer, 2008, pp. 387–396.

[23] S. Thrun, *Explanation-Based Neural Network Learning*, Springer, 1996.

[24] P. Vincent, H. Larochelle, I. Lajoie, Y. Bengio, and P.-A. Manzagol, *Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion*, Journal of Machine Learning Research, 11 (2010), pp. 3371–3408.

[25] R. J. Williams, *Simple statistical gradient-following algorithms for connectionist reinforcement learning*, Machine learning, 8 (1992), pp. 229–256.

[26] T. Zhao, H. Hachiya, G. Niu, and M. Sugiyama, *Analysis and improvement of policy gradient estimation*, in Advances in Neural Information Processing Systems, 2011, pp. 262–270.