Optimal Learning Theory and Approximate Optimal Learning Algorithms

by

Haobei Song

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2019

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

The exploration/exploitation dilemma is a fundamental but often computationally intractable problem in reinforcement learning. The dilemma also impacts data efficiency which can be pivotal when the interactions between the agent and the environment are constrained. Traditional optimal control theory has some notion of objective criterion, such as regret, maximizing which results in optimal exploration and exploitation. This approach has been successful in multi-armed bandit problem but becomes impractical and mostly intractable to compute for multi-state problems. For complex problems with large state space when function approximation is applied, exploration/exploitation during each interaction is in practice generally decided in an ad hoc approach with heavy parameter tuning, such as ϵ-greedy. Inspired by different research communities, optimal learning strives to find the optimal balance between exploration and exploitation by applying principles from optimal control theory.

The contribution of this thesis consists of two parts: 1. to establish a theoretical framework of optimal learning based on reinforcement learning in a stochastic (non-Markovian) decision process and through the lens of optimal learning unify the Bayesian (model-based) reinforcement learning and the partially observable reinforcement learning. 2. to improve existing reinforcement learning algorithms in the optimal learning view and the improved algorithms will be referred to as approximate optimal learning algorithms.

Three classes of approximate optimal learning algorithms are proposed drawing from the following principles respectively:
(1) Approximate Bayesian inference explicitly by training a recurrent neural network entangled with a feed forward neural network;
(2) Approximate Bayesian inference implicitly by training and sampling from a pool of prediction neural networks as dynamics models;
(3) Use memory based recurrent neural network to extract features from observations.
Empirical evidence is provided to show the improvement of the proposed algorithms.

Acknowledgements

I would like to thank my supervisor Mahesh Tripunitara for mentoring me and supporting me, professor Pascal Poupart for introducing me to reinforcement learning research and professor Mark Crowley for his feedback.

Dedication

The thesis is dedicated
to my parents for their unconditional and countless support
to my brother Haonan and wish him success
to my love Agnieszka
to my friends for the time spent at Waterloo

Table of Contents

List of Figures viii

Abbreviations ix

1 Introduction 1

1.1 Learning . . . 1
1.2 Redefinition of Decision Processes . . . 3
1.3 Optimal Learning . . . 4

2 Exploration/Exploitation Dilemma and Data/Computation efficiency 7

2.1 Exploration/exploitation dilemma . . . 7
2.2 Data efficiency/computation efficiency dilemma . . . 16

3 Optimal learning theory 19

3.1 POMDP reduces to SDP-based RL . . . 20
3.2 SDP-based RL reduces to Bayesian model based RL . . . 22
3.3 Bayesian model based RL reduces to POMDP . . . 24

4 Approximate Bayesian inference 26

4.1 Overview of Bayesian inference . . . 26
4.2 Approximate Bayesian inference . . . 30
4.2.1 Bayes-Net and FR-Net . . . 31
4.2.2 Sample from a pool of neural networks . . . 32

5 Approximate optimal learning algorithms and empirical evaluation **37**

5.1 Deep Q learning with information state (S learning) 38

5.2 Model based reinforcement learning with P-Net pool sampling 42

5.3 Stochastic decision process (SDP) based reinforcement learning 46

6 Summary **48**

References **49**

APPENDICES **51**

6.1 Algorithm to generate a random 8×8 maze 51

6.2 Evolutionary algorithm used to train the P-Net pool 55

List of Figures

1.1 Flow graphs of learning 2
1.2 Problem space and classes of reinforcement learning algorithms 4

2.1 $Beta(\alpha, \beta)$ distribution of p_1 after certain number of choosing action a_1 . . 11
2.2 A model based reinforcement learning framework (Dyna) 14

4.1 The belief generated by the Bayes-Net for $t = 2, 3$ 33
4.2 Approximate Bayesian inference. 34

5.1 Architecture of Deep Q learning algorithm with information state (S learning) 38
5.2 Deep Q learning with information state (S learning) averaged over 100 runs of randomly generated maze problems with/without meta-co-training . . . 39
5.3 Deep Q learning with information state (S learning) averaged over 100 runs of randomly generated maze problems. 40
5.4 Architecture of model-based tabular Q learning by sampling neural networks (P-Net) of model dynamics 42
5.5 Model based tabular Q learning with different sampling techniques (averaged over 100 randomly generated maze problems. 43
5.6 Architecture of P-Net for reinforcement learning problems based on visual input (such as video games) 45
5.7 Architecture of a general stochastic decision process based reinforcement learning algorithm 46

Abbreviations

FFN Feed forward neural network 38

GRU Gated recurrent unit 38

LSTM Long short term memory 38

MDP Markov decision process 2

POMDP Partially observable Markov decision process 3

RNN Recurrent neural network 38

SDP Stochastic decision process 3

Chapter 1

Introduction

1.1 Learning

Learning is defined as the interactions between an *environment* and an *agent (controller)*. The interactions include the *state*: a random variable produced by the *environment* and observed by the *agent*, the *action*: a random variable produced by the *agent* which may affect the *environment*, the *reward*, generated from the *environment* passed to the *agent* as a result of the current *state* and *action*. Additionally, *observation* is defined in partially observable reinforcement learning as a random variable which is a function of the *state*. The goal of learning is to find a *policy*

Figure (1.1) illustrates the interactions in learning using a flow diagram. There are two flow paths from environment to agent in learning. One is **information flow** which in the form of observations/states and rewards, contains some information about the model dynamics of the environment and the other is **reward flow** in the form of rewards. There is one **control flow** path from agent to environment through actions. To obtain optimal policy, actions have to be chosen to gather statistically *sufficient amount of information* about the model through information flow, where sufficient amount of information is the least amount of information about the model to obtain optimal policies . To generate the most expected total rewards, actions have to be selected solely for large immediate rewards from reward flow. Therefore, the purpose of the control flow results in the trade-off between benefiting the information flow or the reward flow. In practice, a specific action during each interaction is selected generally for a mixture of immediate reward and gathering of model information to gain long-term reward. Quantifying this long-term reward exactly in the same metric as the immediate reward is generally computationally intractable for

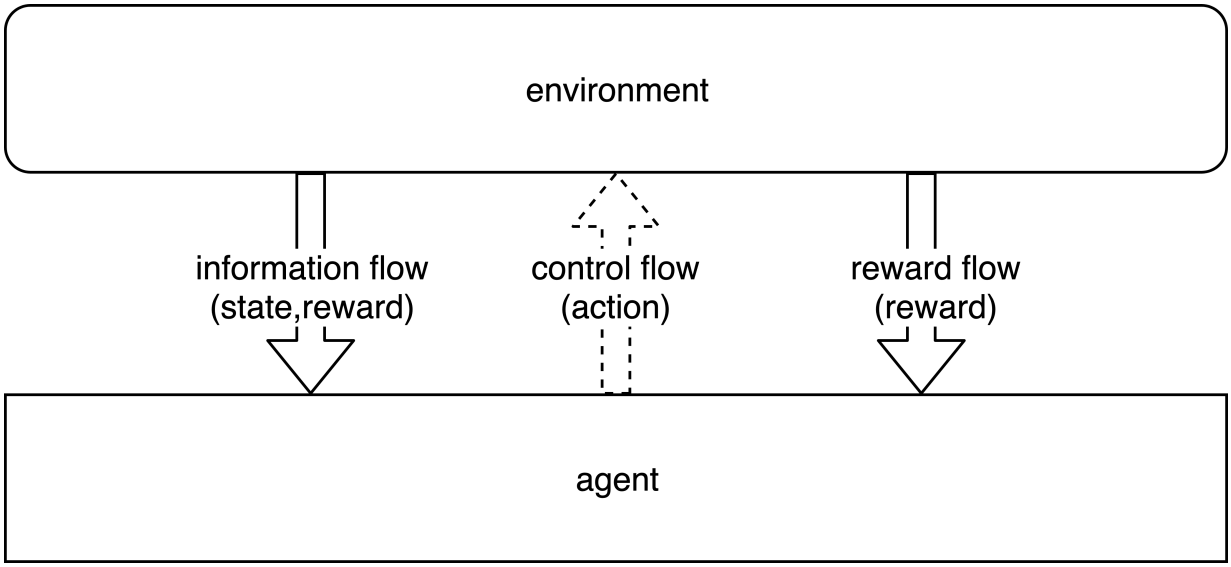


Figure 1.1: Flow graphs of learning

multi-state problems, which results in the exploration/exploitation dilemma as otherwise the optimal action is simply the one with the largest immediate reward plus long term reward.

The learning framework in most of the reinforcement learning study is based on the assumption that the environment can be modelled as a Markov decision process (MDP) and the optimal action selection policy reduces to the problem of solving the following Bellman equation[1]:

$$V^*(s) = \max_a \{\sum_{s',r} P(s', r|s, a) [r + \gamma V^*(s')]\} \tag{1.1}$$

where $V^*(s)$ is an optimal value function of state s, a is action, r is reward, s' is the next following state, $P(s', r|s, a)$ is the transition probability function and γ is the discount factor. Detailed specification of the notations used in the thesis is provide on the next page.

1.2 Redefinition of Decision Processes

In recent years Partially observable Markov decision process (POMDP) as a more complicated formulation, has been widely studied to solve non-stationary (or partially observable)[i] learning problems[11]. In this thesis a general form of sequential decision-making process, named Stochastic decision process (SDP) is defined below to facilitate discussion on non-Markov decision process with MDP and POMDP formulated as special instances in such framework. A *policy* π of an agent is an action selection strategy based on the states in an SDP or the observations in a POMDP.

Stochastic decision process (SDP) An SDP is defined as a tuple $\langle \mathcal{S}, \mathcal{A}, P, P_0 \rangle$, where $\mathcal{S}$ is the state space and $\mathcal{A}$ is the action space. The state $s_{n+1} \in \mathcal{S}$ and reward $r_{n+1} \in \mathbb{R}$[ii] at time step $(n+1)$ follow the probability distribution $P(s_{n+1}, r_{n+1} | s_0, s_1, \cdots, s_n, a_n)$ where $a_n \in \mathcal{A}$ with initial state $s_0 \in \mathcal{S}, s_0 \sim P_0(s_0)$. A policy $\pi(a_n | s_0, s_1, \cdots, s_n)$ for an SDP is typically defined as a probabilistic distribution over actions conditioned on all the past states (or a function of past states for a deterministic policy).

MDP $\langle \mathcal{S}, \mathcal{A}, P, P_0 \rangle$ is an SDP with the Markovian property:

$$P(s_{n+1}, r_{n+1} | s_0, s_1, \cdots, s_n, a_n) = P(s_{n+1}, r_{n+1} | s_n, a_n)$$

A policy $\pi(a_n | s_n)$ for an MDP is thus simplified as a probabilistic distribution of only the current state. A stationary MDP is an MDP such that $P_t(s, r) = P_{t+1}(s, r)$ for any t, s and r while in a non-stationary MDP there is no such constraint.

POMDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{O}, P, P_0 \rangle$ is an MDP with the addition of an observation space $\mathcal{O}$ and probability distribution Ω, where an observation at time step t $o_t \in \mathcal{O}, o_t \sim \Omega(o_t | s_t, a_t)$. The agent is given the state s in an MDP setting where in POMDP the agent cannot access the state but instead the observation is provided. A policy $\pi(a_n | l_n, o_n)$ for a POMDP , where l_n is a *latent* variable defined as a function of the previous observation o_{n-1} and the agent's previous latent variable l_{n-1}, is a function of the agent's current latent variable and observation.

N.B. The notation above draws mainly from Ghavamzadeh et al.[6] and Sutton and Barto[16].

[i] The equivalence between the two can be found in most introductory books on stochastic processes or Markov processes.

[ii] the set of real numbers

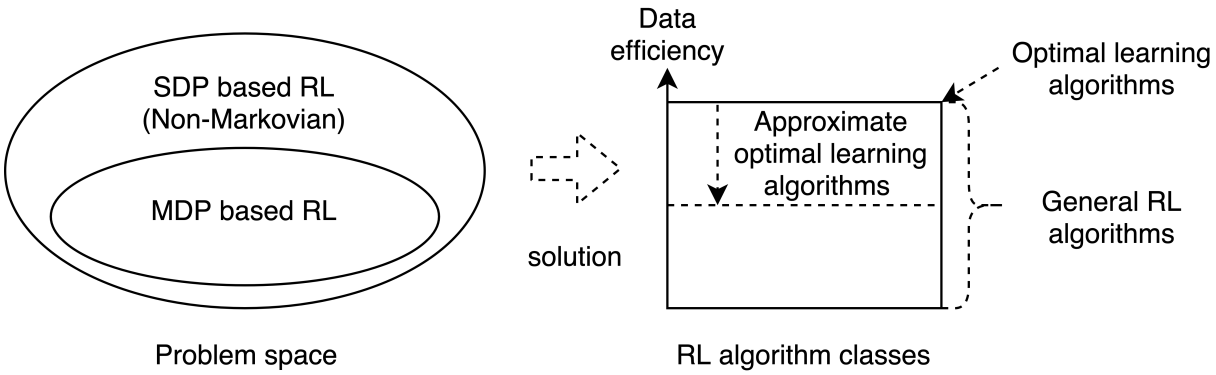


Figure 1.2: Problem space and classes of reinforcement learning algorithms

1.3 Optimal Learning

This thesis attempts to identify the trend in RL research community that more and more research is conducted towards non-Markovian decision processes or SDPs as is defined previously. This trend is likely to have been driven by tackling the exploration/exploitation dilemma or data efficiency as a result of it in RL community. e.g. from MDP to POMDP, from RL to Bayesian model based RL, from simple single state prediction neural network to recurrent neural network, etc. Instead of patching up the existing algorithms in an MDP-based RL formulation to suit non-Markovian models, this thesis formulates problems in the more general SDPs and constructs optimal learning algorithms which perform optimal exploration and exploitation and thus achieves maximum data efficiency. Though the computation of such optimal learning algorithms is mostly intractable, analysis of these theoretical algorithms complements comprehensive understanding of the exploration/exploitation dilemma which would rather be difficult to deal with in an MDP formulation. The optimal learning algorithms will later in the thesis direct the construction of approximate optimal learning algorithms which are RL algorithms with generally improved data efficiency in SDP formulations. The relation between the two problem formulations based on different decision processes and among different classes of algorithms is illustrated in Figure 1.2

Optimal learning is a pattern of interactions which maximizes the expected total reward for all the interactions incurred throughout the entire duration of learning[4].
Reinforcement learning is concerned with how agents ought to take actions in an environment so as to maximize some notion of cumulative reward[19].
Optimal learning can be understood as a subset of reinforcement learning emphasized on

(often theoretically) obtaining optimal policies by performing optimal exploration/exploitation. RL research has been done largely on tasks with effortless access to infinite amount of training data such as computer games, board games (which can be simulated on computers), etc. For problems with constrained interactions with the environment, data efficient RL or OL is in demand. As an example, RL in robotics must take into consideration the worn-out cost of a test robot and testing a new drug might end a patient's life. The reason that OL gains little interest in learning is that OL algorithms are in general intractable to compute. Even for the single state RL problem known as multi-armed bandit, it was not until 27 years later since its formulation by Robbins[13] that a closed form optimal solution named Gittins[7] index was given. Nowadays, almost all the challenging RL problems to solve is computationally intractable if formulated as OL problems and what makes it worse is that even an approximate OL framework is barely available. This thesis was originated as an attempt to give a general-form approximate OL algorithm evolved from a model based RL framework under Bayesian inference with another two approximate OL algorithms proposed along the study.

A brief history of optimal learning Optimal learning has been studied over a few decades and dated back to as early as 1947 in Wald[17]'s initial work of sequential analysis. Over the years, optimal learning has been studied under different names such as "adaptive control process", "dual control" in the control literature. The initial work on model based Bayesian Reinforcement Learning was published under the topic dual control[5]. The central problem is formulated as a sequential decision process under uncertainty and an agent is an algorithm which at each step takes an action in the uncertain environment and changes the state of the environment.

The rest of the thesis starts in Chapter 2 by looking at the exploration/exploitation dilemma, which has perplexed researchers for decades without a computationally tractable solution in general. The trade-off between data and computation efficiency is also highlighted in that chapter under OL frameworks which will be useful later for constructing approximate OL algorithms. Chapter 3 compares variations of non-Markovian RL formulations: POMDP, Bayesian RL and SDP based RL and an informal proof by reduction is provided for proving the uniform optimality of RL algorithms among these RL variations. Chapter 4 reviews the Bayesian statistics in the context of RL and provides a road map to approximate Bayesian inference explicitly and implicitly using neural networks. Approximate Bayesian inference will serve at the main tool to construct and analyze approximate OL algorithms in Chapter 5. Chapter 5 provides details on implementing three classes of approximate OL algorithms under three different RL frameworks and empirical evaluation is shown to align with theoretical analysis. Chapter 6 then sums up the thesis and gives

take-aways.

Chapter 2

Exploration/Exploitation Dilemma and Data/Computation efficiency

2.1 Exploration/exploitation dilemma

Exploration refers to the actions through which the agent aims to infer information about the environment dynamics and if such information is processed correctly it can obtain the most expected long term rewards.

Exploitation refers to the actions selected to maximize the expected immediate rewards in one episode for episodic tasks (or in the limit for continuous tasks) by making use of the present information about the model.

In theory, an action can be selected for the purpose of both exploration and exploitation. But there is always a trade-off between the two as one has to give up on one to benefit the other. The ultimate goal of reinforcement learning is, through interactions with the environment, to obtain a policy which results in the most amount of expected rewards in one episode for episodic tasks or in the limit for continuous tasks. To achieve this goal in theory, it is sufficient for general RL algorithms to obtain implicitly (model free RL) or explicitly (model based RL) statistically sufficient information about the model in the limit of infinite number of interactions. In practice, the asymptotic convergence guarantee adapts effectively to algorithms with large number of interactions such as Atari games, Go, etc. However, in many problems such as drug test, robot control, etc., intermediate rewards under a sub-optimal policy are far more important than an optimal policy trained

from a large amount of interactions[3]. Therefore, an OL algorithm is often in demand to solve the problems above as an optimal policy deduced from an OL algorithm must balance the optimal trade-off between exploration and exploitation and consequently achieve the optimal data efficiency. But OL algorithms only exist in theory for most problems with a large state space and are generally intractable to compute in practice.

There has been research on explicitly quantifying the value of an action (at certain state) with respect to obtaining more information about the model and obtaining more immediate rewards. UCB(upper confidence bound) and Thompson sampling were proposed to address the exploration/exploitation dilemma quantitatively for multi-armed bandit problems. Their effectiveness can be well explained and understood in an MDP-based RL with Bayesian Prior being the parameters of Dirichlet distributions. For multi-state RL problems, it is often computationally intractable to compute the value functions.

Data efficiency is closely related to exploration/exploitation dilemma and describes the amount of training data required to obtain an optimal policy in the context of RL. In practice, the concept of data efficiency is extended RL algorithms that obtain only suboptimal policies. This extension adds to the complication of data efficiency analysis as the following situation has to be addressed where some RL agent can learn a suboptimal policy with little training data while another RL agent might be able to obtain a better policy with more data. Intuitively, we want to quantify the 'true' data efficiency for policies of different optimality, and it should be in such form as the value function of a policy divided by the amount of training data required to learn such policy. Unfortunately, there is no closed form criterion that both captures the essence of data efficiency and is computationally feasible for problems of large state space especially when function approximation is applied. Perfect exploration obtains sufficient amount of information of the model dynamics for reasoning an optimal policy and perfect exploitation results in the best effort expected total rewards based on the limited information about the model dynamics learned from the previous exploration. Improving data efficiency is about quantitatively balancing out exploration and exploitation to obtain the most expected total rewards with a limited number of interactions (or in a finite horizon).

In theory, optimal learning has some closed form notion of criterion (such as regret in bandit problems) maximizing which results in optimal data efficiency. As will be shown in the next chapter, Bayesian model based RL is equivalent to an optimal learning formulation. A closer look at the Bayesian model based RL algorithm suggests that the quantitative evaluation of exploration/exploitation for each action at a time can be dealt with by quantifying the uncertainty of the model dynamics and then computing the expected future rewards conditioned on the uncertainty in the model. Under such reasoning,

pure exploration corresponds to infinite uncertainty in the state value so the agent will take a random action and pure exploitation means there is no uncertainty in the perceived model dynamics so the agent has learned the true model and will strictly follow the optimal policy deduced from the model.

Here I give a 'hello world' example optimal learning algorithm which is also referred to as Bayesian model based RL algorithm to solve the classic *stationary two-armed bandit problem* and analyze the exploration/exploitation dilemma as well as the data efficiency.

Stationary two-armed bandit problem is formulated as two slot machines corresponding to two actions $a \in \{a_1, a_2\}$ where a_1, a_2 means choosing the slot machine 1 and 2 respectively and an agent has some probability of hitting the jackpot of the selected slot machine. 'Stationary' specifies the slot machines are fixed so the state in the problem is trivial and does not change over time. During interactions, the agent selects an action and the corresponding slot machine generates rewards $r \in \{1, 0\}$ according to a Bernoulli distribution with an unknown but fixed probability p_1, p_2 (p_1, p_2 are generated from a uniform distribution $\mathcal{U}(0, 1)$ before the experiment starts) for each slot machine (a_1, a_2) respectively. The goal of an optimal learning agent is to maximize the expected total rewards within a specified number of interactions (finite horizon).

N.B. Obviously, once the value of p_1, p_2 are known, the optimal action is to simply choose the slot machine with larger p. Therefore, the agent needs to (1) explore and choose both actions in order to infer the true parameters p_1, p_2; (2) exploit the information gained from exploration and always choose the action with larger $p_i, i = 1, 2$. Though the values of p_1, p_2 are totally unknown, there is *a priori* knowledge about them in the problem described above, for they are uniformly generated at the start. The uncertainty in the belief of the value of p_1, p_2 complicates the analysis as in fact one needs to compare two random variables p_1, p_2 with their uncertainty in consideration.

Bayesian model based reinforcement learning In Bayesian statistics, the belief about the parameters p_1, p_2 has to be constantly updated. This belief is also referred to as Prior/Posterior before/after updating. The belief describes the probability distributions of the random variables p_1, p_2 after every interaction. Here, $Beta(\alpha_1, \beta_1)$ and $Beta(\alpha_2, \beta_2)$ distributions are used to encode our belief about p_1 and p_2 and $\alpha_1, \beta_1, \alpha_2, \beta_2$ are hyperparameters which parametrize the distributions of the random variables. The usage of the $Beta$ distribution has some mathematical evidence, e.g. the mean of a $Beta$ distribution coincides with the sample mean.

the probability density function of $Beta(\alpha, \beta)$ is

$$
\begin{aligned}
f(p; \alpha, \beta) &= \frac{\Gamma(\alpha+\beta)p^{\alpha-1}(1-p)^{\beta-1}}{\Gamma(\alpha)\Gamma(\beta)} \\
mean &= \int_p \frac{\Gamma(\alpha+\beta)p^{\alpha-1}(1-p)^{\beta-1}}{\Gamma(\alpha)\Gamma(\beta)} p dp \\
&= \int_p \frac{(\alpha+\beta)\alpha\Gamma(\alpha+\beta)p^{\alpha}(1-p)^{\beta-1}}{(\alpha+\beta)\alpha\Gamma(\alpha)\Gamma(\beta)} dp \\
&= \int_p \frac{\alpha\Gamma(\alpha+\beta+1)p^{\alpha}(1-p)^{\beta-1}}{(\alpha+\beta)\Gamma(\alpha+1)\Gamma(\beta)} dp \\
&= \frac{\alpha}{\alpha+\beta} \int_p \frac{\Gamma(\alpha+\beta+1)p^{\alpha}(1-p)^{\beta-1}}{\Gamma(\alpha+1)\Gamma(\beta)} dp \\
&= \frac{\alpha}{\alpha+\beta} \times 1 = \frac{\alpha}{\alpha+\beta}
\end{aligned}
$$

where $\Gamma(x) = (x-1)!, x \in \mathbf{N}^+$

The initial values of hyper-parameters are all 1's making $Beta(1, 1)$ a uniform distribution, which intuitively corresponds to the fact that before any training incurred the agent has absolutely no knowledge about the parameters p_1, p_2 except that they are generated uniformly. During the training, the hyper-parameters are updated every interaction and the distribution would be skewed to the true value of p_1, p_2 if learned properly. The agent then chooses the optimal action based on the belief of the model, which is a distribution of the true parameters of model dynamics. The uncertainty in the belief distribution instructs the agent to perform optimal exploration at each interaction.

Start with $\alpha_1 = 1, \beta_1 = 1, \alpha_2 = 1, \beta_2 = 1$ and denote the estimates of p_1, p_2 as P_1, P_2, two random variables following probability distributions $Beta(\alpha_1, \beta_1), Beta(\alpha_2, \beta_2)$ respectively. During the initial interaction, the agent has equal probability of choosing action a_1 or a_2 as $Beta(1, 1)$ is a uniform distribution. The belief will be updated based on the reward received after that action is taken using Bayesian inference. Since $Beta$ distribution belongs to the conjugate distribution family, the Posterior also follows $Beta$ distribution as is proved in Equation 2.1.

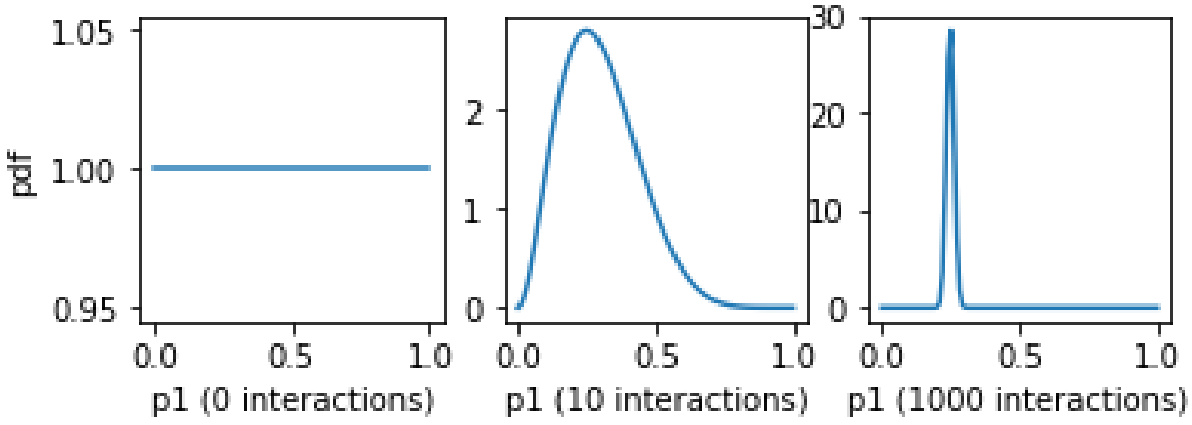


Figure 2.1: $Beta(\alpha, \beta)$ distribution of p_1 after certain number of choosing action a_1

Let $f(p_1; \alpha_1, \beta_1)$ be the probability density function of $Beta(\alpha_1, \beta_1)$ distribution, R a random variable of the rewards and A a random variable of actions, it follows:

$$Prob(P_1 = p_1 | R = 1, A = a_1) = \frac{Prior(P_1 = p_1) Prob(R = 1 | P_1 = p_1)}{\int_{p'} Prob(P_1 = p') Prob(R = 1 | P_1 = p'_1)} dp'_1 \tag{2.1}$$

$$= \frac{f(p_1; \alpha_1, \beta_1) p_1}{\int_{p'_1} f(p'_1; \alpha_1, \beta_1) p'_1 dp'_1} \tag{2.2}$$

Note the denominator is the mean of $Beta(\alpha_1, \beta_1)$ distribution which is $\frac{\alpha_1}{\alpha_1 + \beta_1}$

$$= \frac{\frac{\Gamma(\alpha_1 + \beta_1) p_1^{\alpha_1 - 1} p_1 (1 - p_1)^{\beta_1 - 1}}{\Gamma(\alpha_1)\Gamma(\beta_1)}}{\frac{\alpha_1}{\alpha_1 + \beta_1}} \tag{2.3}$$

$$= \frac{\Gamma(\alpha_1 + \beta_1 + 1) p_1^{\alpha_1} (1 - p_1)^{\beta_1}}{\Gamma(\alpha_1 + 1)\Gamma(\beta_1)} \tag{2.4}$$

$$= f(p_1; \alpha_1 + 1, \beta_1) \tag{2.5}$$

similarly

$$Prob(P_1 = p_1 | R = 0) = f(p_1; \alpha_1, \beta_1 + 1) \tag{2.6}$$

If the reward is 1 after taking action a_1, α_1 will be incremented by 1 with β_1 remaining the same and vice versa if the reward is 0 with α_2, β_2 remaining the same. Figure 2.1 is

the distribution belief of p_1 (probability density function of P_1) after choosing a_1 a certain number of times with the true parameter $p_1 = 0.25$. Let ΔP be a random variable of the difference between P_1 and P_2 that $\Delta P = P_1 - P_2$. At an arbitrary time step, one induced policy π can be defined as: (A is a random variable of action)

$$\pi(A = a_1) = Prob(\Delta P > 0) = g(\alpha_1, \beta_1, \alpha_2, \beta_2) \tag{2.7}$$
$$\pi(A = a_2) = 1 - \pi(A = a_1) = g(\alpha_2, \beta_2, \alpha_1, \beta_1) \tag{2.8}$$

$g(\alpha, \beta, \alpha', \beta')$ is a function mapping to the quantity of $Prob(\Delta P > 0)$. Let

$$h = \frac{B(a + c, b + d)}{B(a, b)B(c, d)}$$

where

$$B(\alpha, \beta) = \frac{\Gamma(\alpha)\Gamma(\beta)}{\Gamma(\alpha + \beta)}$$

is the normalization term for $Beta$ distribution that

$$f(p; x, y) = \frac{p^{\alpha-1}(1 - p)^{\beta-1}}{B(x, y)}$$

A recursive method to calculate $g(p; \alpha, \beta, \alpha', \beta')$ is shown by Cook[2]:

$$\begin{aligned}
g(\alpha_1 + 1, \beta_1, \alpha_2, \beta_2) &= g(\alpha_1, \beta_1, \alpha_2, \beta_2) + h(\alpha_1, \beta_1, \alpha_2, \beta_2)/a \\
g(\alpha_1, \beta_1 + 1, \alpha_2, \beta_2) &= g(\alpha_1, \beta_1, \alpha_2, \beta_2) - h(\alpha_1, \beta_1, \alpha_2, \beta_2)/b \\
g(\alpha_1, \beta_1, \alpha_2 + 1, \beta_2) &= g(\alpha_1, \beta_1, \alpha_2, \beta_2) - h(\alpha_1, \beta_1, \alpha_2, \beta_2)/c \\
g(\alpha_1, \beta_1, \alpha_2, \beta_2 + 1) &= g(\alpha_1, \beta_1, \alpha_2, \beta_2) + h(\alpha_1, \beta_1, \alpha_2, \beta_2)/d
\end{aligned}$$

with

$$g(\alpha_1, \beta_1, \alpha_1, \beta_1) = \frac{1}{2}$$

It can be proved that the agent obtains the most expected total rewards with policy π defined in Equation 2.7 and 2.8 assuming P_1 and P_2 follow $Beta$ distributions. If instead sampling is used to estimate the probability of $Prob(\Delta P) > 0$, this approach becomes Thompson sampling. This optimal learning algorithm can be obviously extended to problems with large discrete action and reward space using $Dirichlet$ distribution to model the probability of obtaining a specific reward for each action. In general, this approach can be applied to any RL problems when the transition function $P(r, s'|s, a)$ can be parameterized

by independent parameters or in other words all the elements in the *generalized transition matrix* are independent. Let $p^a_{i,j}$ be the corresponding probability of transition from state i to j when taking action a. A generalized transition matrix for reward r is of the following form.

$$P(r) = \begin{bmatrix} p^1_{1,1} & p^1_{1,2} & \cdots & p^1_{1,|S|} \\ p^2_{1,1} & p^2_{1,2} & \cdots & p^2_{1,|S|} \\ \vdots & \vdots & \vdots & \vdots \\ p^{|A|}_{1,1} & p^{|A|}_{1,2} & \cdots & p^{|A|}_{1,|S|} \\ p^1_{2,1} & p^1_{2,2} & \cdots & p^1_{2,|S|} \\ p^2_{2,1} & p^2_{2,2} & \cdots & p^2_{2,|S|} \\ \vdots & \vdots & \vdots & \vdots \\ p^{|A|}_{2,1} & p^{|A|}_{2,2} & \cdots & p^{|A|}_{2,|S|} \\ \vdots & \vdots & \vdots & \vdots \\ p^1_{|S|,1} & p^1_{|S|,2} & \cdots & p^1_{|S|,|S|} \\ p^2_{|S|,1} & p^2_{|S|,2} & \cdots & p^2_{|S|,|S|} \\ \vdots & \vdots & \vdots & \vdots \\ p^{|A|}_{|S|,1} & p^{|A|}_{|S|,2} & \cdots & p^{|A|}_{|S|,|S|} \end{bmatrix} (r)$$

All the $p_{i,j}$'s are assumed to be mutually independent and the problem to be solved is a stationary Markov decision process with finite sets of states, actions and rewards. Then for each (state, next state, action) triplet,

$$\{p^k_{i,j}(r) | r \in R, R \text{ is a discrete set of rewards}\}$$

is a set of parameters to specify the probability distribution of obtaining reward r throughout interactions. This set of parameters can be hyper-parametrized using a *Dirichlet* distribution with $|R|$ number of hyper-parameters. The collection of hyper-parameters for all the parameters in P can thus serve as the belief in an optimal learning algorithm. The optimal learning algorithm for stationary two-armed bandit falls under such diagram with (where subscript 0 represents the only default state):

$$P(r=0) = \begin{bmatrix} p^1_{0,0}(r=0) \\ p^2_{0,0}(r=0) \end{bmatrix}, P(r=1) = \begin{bmatrix} p^1_{0,0}(r=1) \\ p^2_{0,0}(r=1) \end{bmatrix} \tag{2.9}$$

Therefore, the algorithm would need two sets of hyper-parameters to describe the two *Dirichlet* distributions for both actions in $\{a_1, a_2\}$ (with same default state). Each set of hyper-parameters has cardinality 2 with *Beta* distribution being the special case of *Dirichlet* distribution when there are two different rewards).

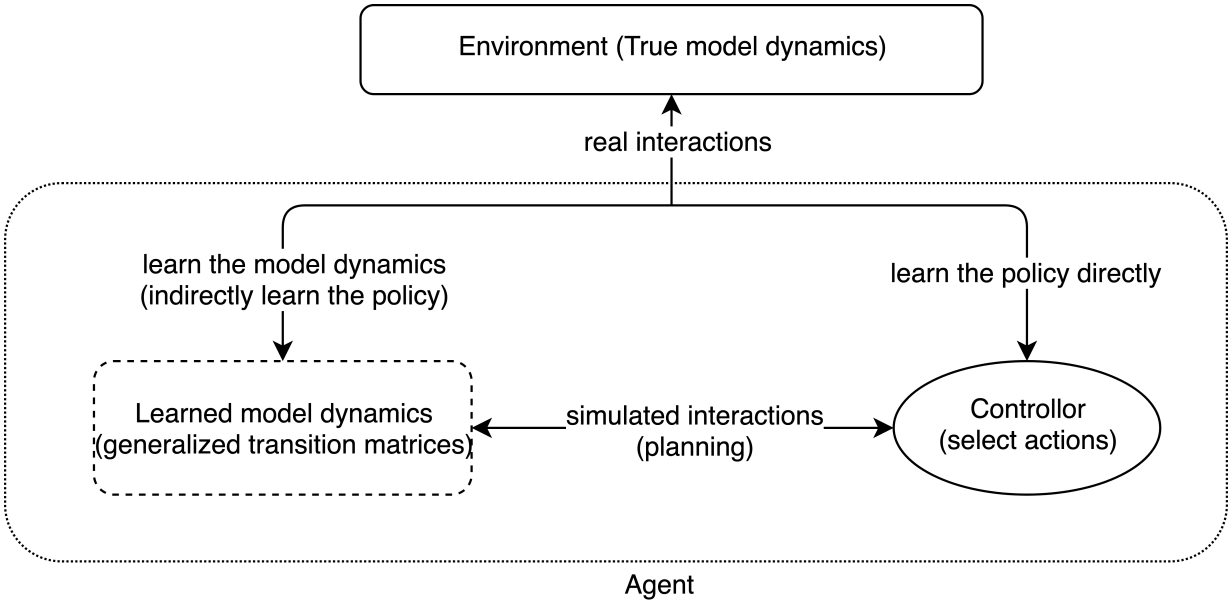


Figure 2.2: A model based reinforcement learning framework (Dyna)

From my knowledge no non-recurrent solution is known by applying Bayesian optimal learning algorithm above to single state two armed bandit problem due to the computation of $g(\alpha, \beta, \alpha', \beta')$. For multi-state problems, the Bayesian approach with *Dirichlet* distribution as belief model would require $|S|^2|A||R|$ (R is reward set) hyper-parameters and the belief update rule is overly complicated and often computationally intractable. What makes it even worse is that no general *approximate* OL algorithm is available . The following chapter will give three different but equivalent theoretically optimal learning formulations and approximate optimal learning algorithms under such frameworks will be constructed and analyzed thereafter.

Brief introduction to model based reinforcement learning There has been an increasing amount of research in data efficient reinforcement learning and among which model based reinforcement learning has succeeded in several benchmark problems such as inverted pendulum. Figure 2.2 is the architecture of a model based RL algorithm named *Dyna* [15]. In general, a model based RL agent learns a policy and in addition a dynamics model (also from the real interactions) which represents some knowledge learned from the past interactions about the true environment. At each time step, the agent learns from the real interactions with the true environment and simulated interactions generated from the

perceived model dynamics (or possibly learn from the model dynamics directly, e.g., learn from transition matrices directly for problems with small state space by value function *sweeping*). Updating the dynamics model and learning indirectly from the learned model is generally referred to as *planning*, which often requires a large amount of computation power and becomes the bottleneck of model-based RL. From the information flow diagram of Figure 1.1, learning from the dynamics model is redundant as the model itself is learned from the real interactions. However, in a reinforcement learning context, such redundancy is important as the learning algorithm is not perfect at each interaction. The agent needs to 'look back' at a later time and might learn something different since the 'context' for the agent has changed after it has learned something new in between.

With appropriate control over the balance between learning directly from real interactions and indirectly from its learnt dynamics model, more computation power allocated to the planning can mostly result in better data efficiency (with respect to the amount of real interactions needed). This has brought up another dilemma in reinforcement learning, namely, computation efficiency vs. data efficiency.

2.2 Data efficiency/computation efficiency dilemma

Due to the limited computational resources in practice, exact computation of optimal learning is in general computationally intractable for multi-state problems. As an example planning till convergence in Bayesian model based RL such as computing $g(\alpha, \beta, \alpha', \beta')$ in the example problem in the previous section is not always possible and the quality of a learned policy can therefore be traded for the amount of assigned computation resources and better policies generally require more computation when other conditions are fixed.

Computation efficiency describes the amount of computation power required to learn or derive an optimal policy in reinforcement learning formulation. Like data efficiency, computation efficiency extends to learning a suboptimal policy, but the definition is not clear in RL formulation as there is no agreed metric to evaluate the quality of a suboptimal policy with respect to the amount of computation resources in use. Intuitively, Computation efficiency can be quantified as the optimality of a policy divided by the amount of computational resources supplied. There is so far no general notation to relate an optimal learning algorithm to its practical computational complexity. It makes sense as OL traditionally aims primarily at finding the optimal solution regardless of the computational complexity. Nevertheless, OL formulation facilitates the understanding of how computation efficiency relates to data efficiency with unlimited computation able to achieve optimal data efficiency. In practice, computational complexity is important because model-based reinforcement learning has to trade off between data efficiency and computational efficiency when planning becomes the bottleneck. Approximate optimal learning is proposed to direct the approximation of computing an optimal policy to sacrifice minimum data efficiency. Later I will show that in fact, the difficulty of planning in Bayesian model based RL relates to the computational intractability of an OL formulation. e.g. the optimal learning algorithm for stationary two-armed bandit problem has to compute $g(\alpha, \beta, \alpha', \beta')$, which corresponds to the planning in a Bayesian model based RL.

In general, data efficiency can be traded with computation efficiency in a Bayesian model based reinforcement learning framework, since more computation on planning can complement the lack of training data. Similar to exploration/exploitation, an approximate algorithm with sub-optimal data efficiency is often better than algorithms with optimal data efficiency which however requires tremendous computational power. To rewind, the main purpose of model based reinforcement learning is to improve data efficiency partly due to exploration/exploitation dilemma, but this unfortunately leads to another dilemma of data vs. computation efficiency.

Approximate optimal learning Exact optimal learning algorithm is generally more difficult to derive and implement compared to general RL algorithm. As is shown previously, even for the simple stationary two-armed bandit problem, there is no obvious non-recurrent solution. In practice, an approximate optimal learning solution is often sufficient for our demand. Interestingly, the idea of approximate optimal learning collides with the Bayesian model based reinforcement learning in many aspects as will be shown in the next chapter that they are equivalent. The following is an example of approximating an optimal learning algorithm, in this case to approximate Bayesian reinforcement learning algorithm by sampling.

Continue with the previous stationary single state two-armed bandit problem to derive an approximate optimal learning algorithm. The optimal policy at each time step is given in Equation 2.7, 2.8. Now we want to obtain an 'approximately optimal policy' π' as

$$\pi'(A = a_1) \approx \pi(A = a_1) = Prob(\Delta P > 0) = g(\alpha_1, \beta_1, \alpha_2, \beta_2)$$

Instead of computing the exact probability of $Prob(\Delta P > 0)$ inevitably in a recurrent approach, we will approximate $Prob(\Delta P > 0) = Prob(P_1 - P_2 > 0) = Prob(P_1 > P_2)$ using Monte Carlo method. Given the assumption that P_1 and P_2 follow $Beta$ distributions, one can sample p_1, p_2 from the corresponding $Beta$ distribution with the maintained hyper-parameters $\alpha_1, \beta_1, \alpha_2, \beta_2$. The algorithm is given below with the sampling technique often referred to as Thompson sampling in model-based reinforcement learning literature.

Algorithm 1: Approximate optimal learning by sampling
(equivalent to a Bayesian model-based RL with Thompson sampling)

Result: Compute π'
Input: $state = 0, action(A) \in \{a_1, a_2\}, reward \in \{0, 1\}$
Initialize $\alpha_1 = \beta_1 = \alpha_2 = \beta_2 = 1$;
while $\max\{\alpha_1, \beta_1, \alpha_2, \beta_2\}$ *does not reach some threshold* **do**
  if $A = a_1$ **then**
    $\alpha_1 \leftarrow \alpha_1 + reward, \beta_1 \leftarrow \beta_1 + (1 - reward)$;
  else
    $\alpha_2 \leftarrow \alpha_2 + reward, \beta_2 \leftarrow \beta_2 + (1 - reward)$;
  end
  Sample (p_1, p_2) from $Beta(\alpha_1, \beta_1)$ and $Beta(\alpha_2, \beta_2)$ respectively n times ;
  Record the number of times that $p_1 > p_2$ as m ;
  $\pi'(A = a_1) = \frac{m}{n}$;
  $\pi'(A = a_2) = 1 - \pi'(A = a_1)$;
end

This algorithm can be viewed as a model-based reinforcement learning and the planning is performed in a Bayesian approach with $\alpha_1, \beta_1, \alpha_2, \beta_2$ representing the dynamics model. At each time step, the agent updates the dynamics model by incrementing one of $\alpha_1, \beta_1, \alpha_2, \beta_2$ by 1 and learn the approximate optimal policy by approximate $P_1 > P_2$ by sampling.

Note that the above approximate OL algorithm with Thompson sampling cannot scale to problems with large state space especially when function approximation is used because there is no efficient approach to combine the sampled parameters of the transition dynamics. Theoretically, one needs to solve the policy evaluation equation for each set of sampled model parameters which is itself an instance of RL problem with known transition dynamics. Then the value function can be updated by averaging the resulting value functions for each set of sampled transition parameters. Computing the aggregated value function quickly becomes impractical as the number of states increase.

Chapter 3

Optimal learning theory

Reinforcement learning gives no efficiently computable solution to the *exploration/exploitation* dilemma for problems with large state space. In the picture of optimal learning, exploration/exploitation problem is automatically embodied into the problem formulation as data efficiency is now part of the consideration of choosing an optimal action during each interaction. In fact, the uniform optimality of three different optimal learning formulations is proved hereby: (1) POMDP (2) SDP-based RL and (3) Bayesian model based RL. The proof is given by cyclic reduction and the constructed algorithms use ' ′ ' to differentiate descriptions from the original algorithm with the notation {original environment: E, original state: s, original action: a, original reward: r, original observation: o}, {transformed environment: E', transformed state: s', transformed action: a', transformed reward: r', transformed observation: o'}

3.1 POMDP reduces to SDP-based RL

Proof 3.1.1 (POMDP ⇒ SDP-based RL) *Given a problem formulated as a POMDP* $\langle \mathcal{S}^E, \mathcal{A}^E, \mathcal{O}^E, P^E, P_0^E \rangle$ *and an optimal algorithm to solve the problem denoted by* **ALG1**, *from the original environment* $\mathbf{E}$ *form a new environment* $\mathbf{E}'$ *with the new* **state**′ *produced by* $\mathbf{E}'$ *being the old* **observation** *produced by* $\mathbf{E}$ *and everything else is derived accordingly. Note that additional uncertainty is added to the* ***reward*** r_t *as it depends on the 'true' past states instead of observations and there is possible information loss in the generated observations from states. Construct a reinforcement learning algorithm (Algorithm 2) in a SDP* $\langle \mathcal{S}^{E'}(=\mathcal{O}^E), \mathcal{A}^{E'}(=\mathcal{A}^E), P^{E'}, P_0^{E'} \rangle$ *denoted by* **ALG2** *with the environment being* $\mathbf{E}'$.

Algorithm 2: ALG2 with **ALG1** as sub-routine

Result: Select an action

$t = 1$;

Initialize l'_0 (latent variable) as in **ALG1**;

Input: $s'_1, s'_2, \cdots, s'_t$ (*equivalently* $= o_1, o_2, \cdots, o_t$), a'_t, r'_t;

while $\mathbf{E}'$ *is at non-terminal state* **do**

  Update l'_t the same as in **ALG1** with $(o_t = s'_t, l_{t-1} = l'_{t-1}, a_t = a'_t, r_t = r'_t)$;

  Select action the same way as in **ALG1** using (s'_t, l'_t, a'_t, r'_t);

  t=t+1;

end

Aside from the randomness in true reward function conditioned on true states

$$r_{n+1} \sim \int_{s_{n+1}} P^E(s_{n+1}, r_{n+1} | s_n, a_n) ds_{n+1}$$

there is additional uncertainty due to partial observations in POMDP $\langle \mathcal{S}^E, \mathcal{A}^E, \mathcal{O}^E, P^E, P_0^E \rangle$.

$$r'_{n+1} = \int_{s_{n+1} \in \mathcal{S}^E} P^E(s_{n+1}, r_{n+1} | l_n, o_n(=\Omega(s_n, a_n)), a_n) ds_{n+1} \tag{3.1}$$

$$\approx \int_{s_{n+1}} P^E(s_{n+1}, r_{n+1} | s_n(\approx^{\text{i}}(l_n, o_n)), a_n) ds_{n+1} \tag{3.2}$$

[i] '≈' here is a lenient notation for statistically information sufficiency

The uncertainty is inevitably inherited by E' from the view of its 'fake' states. But since **ALG1** *is optimal with the additional reward uncertainty in consideration,* **ALG2** *must be optimal. As is suggested in the Equation 3.2 the additional uncertainty is addressed by the latent variable l_n in* **ALG1**. *Because* **ALG1** *is arbitrary, optimality in POMDP results in the optimality in SDP-based RL. The insight for the proof above is from the equivalence between non-stationary MDP and partially observable MDP.*

3.2 SDP-based RL reduces to Bayesian model based RL

Proof 3.2.1 (SDP-based RL ⇒ Bayesian model based RL) *Given a problem formulated as SDP-based RL* $\langle \mathcal{S}^E, \mathcal{A}^E, P^E, P_0^E \rangle$ *and an optimal algorithm to solve the problem denoted by* **ALG2**, *find a Bayesian inference rule* ***BayesInf***, *of which with the Prior at any time step, any past states can be generated (a tight bound would be a Bayesian inference rule that passes the least amount of information but still makes optimal action selection). The existence of such Bayesian inference rule is obvious as one can simply store/memorize all the previous states encountered. Definite (small) size of Bayesian belief (Prior/Posterior) is in general necessary to derive differentiable policies regardless of using tabular method or function approximation. The Bayesian belief of definite size however makes either the corresponding Bayesian belief update rule intractable or policy derivation and policy improvement overly complicated. Then from the original environment* $\mathbf{E}$ *form a new environment* $\mathbf{E}'$ *which gives out only current state with everything else derived accordingly. N.B. the reward function*

$$R^{E'}(r_{n+1}|s_n, a_n) \approx \int_{s_{n+1} \in \mathcal{S}^E} P^E(s_{n+1}, r_{n+1}|s_0, s_1, \cdots, s_n, a_n) ds_{n+1} \tag{3.3}$$

is relaxed[ii] *to additionally depend on past states (this will add uncertainty in the resulting reward in the view of only current state). Construct a (most likely model-based) RL algorithm* **ALG3** *(Algorithm 3) in a MDP* $\langle \mathcal{S}^{E'}(=\mathcal{S}^E), \mathcal{A}^{E'}(=\mathcal{A}^{E'}), P^{E'}, P_0^{E'}(=P_0^E) \rangle$ *based on* **ALG2**. *Similarly, optimality in SDP-based RL leads to the optimality in Bayesian model based RL.*

It is not obvious how **ALG3** *addresses the additional uncertainty in* $P^{E'}$ *and* $R^{E'}$ *without access to the past states. Since a MDP-based RL algorithm is in general unable to derive an optimal policy without accessing the past states which evolves according to a stochastic decision process, it must be* ***BayesInf*** *that eradicates the additional uncertainty by encoding the useful (statistically sufficient) information about all the past states in the Prior. As a result, Bayesian model-based reinforcement learning mainly encodes into Bayesian belief the additional uncertainty of the environment dynamics* $P^{E'}(s_{n+1}, r_{n+1}|s_n, a_n)$, *namely the* ***model***. ***Planning*** *is necessary in model-based reinforcement learning and it consists of two parts:*
(1) update the model (or the Bayesian belief)
(2) derive and improve the policy of the agent

[ii] This relaxation is stronger than the reward defined in MDP

Algorithm 3: ALG3 with **ALG2** as sub-routine

Result: Select an action

1 $t = 1$;
2 Initialize **Prior** by **BayesInf**;
3 **Input:** s'_t *(no access to previous states at any time step)*,a'_t, r'_t
4 **while** $\mathbf{E}'$ *is at non-terminal state* **do**
5 **if** $t \neq 1$ **then**
6 Generate all the past states $s_1, s_2, \cdots, s_{t-1}$ using **BayesInf**;
7 **end**
8 Compute the **Posterior** from (**Prior**, s'_t, a'_t, r'_t) by **ALG2**;
9 **Prior** $\leftarrow$ **Posterior**;
10 Let $s_t = s'_t$;
11 Select an action using **ALG2** from $(s_1, s_2, \cdots, s_t, a'_t, r'_t)$;
12 t=t+1;
13 **end**

As a comparison between Bayesian model based RL and Bayesian model based RL, what planning does in Bayesian model based RL corresponds to line 5 *and* 7*. Most of model-based RL study focuses on parametric model, but the planning of which is difficult*[iii] *and often intractable for even slightly complicated problems because 'planning occurs in information space', quoted from LaValle*[10]*, where an* ***information state*** *can be interpreted as the observed state augmented with Bayesian belief. To circumvent massive computation spent on updating the model, conjugate families of probability distribution have been proposed to model the uncertainty in the model of the environment*[14]*. But the fundamental computational complexity is still there and it simply shifts the burden from updating the model to reasoning about the model and improving the policy. In chapter 4 an algorithm on approximate Bayesian model-based RL is given which paves the way for solving complicated problems practically by performing planning implicitly.*

[iii] Multi-armed bandit problem is one of the 'simplest' problems that uses parametric model to quantify the uncertainty in the reward function (with only one state)

3.3 Bayesian model based RL reduces to POMDP

Proof 3.3.1 (Bayesian model-based RL ⇒ POMDP) *Given a problem formulated as MDP-based RL and a Bayesian inference system* ***BayesInf*** *with definite size of Bayesian belief, with which an optimal algorithm to solve the problem is denoted by* **ALG3**, *a Bayesian model based RL, then from the original environment* $\mathbf{E}$ *form a new environment* $\mathbf{E}'$ *which generates observations* $o'_t \in \mathcal{S}$ *being states in* $\mathbf{E}$ *and everything else is defined accordingly. Construct an algorithm* **ALG1** *(Algorithm 4) in this POMDP with* **ALG3** *as subroutine. To stretch the proof, the Bayesian belief (Prior) of* ***BayesInf*** *can be thought as latent variable* l_t *in a POMDP and the value of it at time step t (Posterior) is updated from its previous value (Prior) following Bayesian belief update rule in* ***BayesInf***. *If* **ALG3** *is an equivalent Bayesian model-based RL, line 5 in Algorithm 4 would corresponds to the update of the model as part of planning and policy derivation and improvement correspond to line 8. Similarly, the optimality in Bayesian model based RL results in the optimality in*

Algorithm 4: ALG1 with **ALG3** as sub-routine

Result: Select an action
1 **Input:** $o'_t(= s_t)$ *(no access to previous states at any time step)*,a'_t, r'_t;
2 $t = 1$;
3 Initialize latent variable l_0 as the **Prior** from **BayesInf**;
4 **while** $\mathbf{E}'$ *is at non-terminal state* **do**
5 Compute the **Posterior** from (**Prior** $l_{t-1}, o'_t, a'_t, r'_t$) using **BayesInf**;
6 **Prior**← **Posterior**;
7 $l_t \leftarrow$ **Prior**;
8 Select action using **ALG3** with (o'_t, l_t, a'_t, r'_t);
9 t=t+1;
10 **end**

POMDP. The uncertainty in the model can be statistically efficiently specified by Bayesian belief, which if taken as a latent variable in a POMDP is sufficient to derive an optimal policy. However, updating latent variable and utilizing such information to improve the policy is as difficult as planning in a Bayesian approach in model-based RL. The intuition is that the Bayesian belief in Bayesian model based RL can be interpreted as the latent variable in POMDP.

This completes the proof of the three equivalent optimal learning formulations by cyclic reduction. The reasoning in this chapter is not a formal mathematical proof but instead

given to solely address the trend in reinforcement learning community that more and more research has been working its way through towards optimal learning by relaxing the problem formulation to a broader class of problems. The driving force is to tackle the exploration/exploitation dilemma which has perplexed RL researchers for a long time. Examples can be from MDP to POMDP, from traditional RL to Bayesian model-based RL, from simple single state prediction neural network to recurrent neural network, etc.

Chapter 4

Approximate Bayesian inference

4.1 Overview of Bayesian inference

The probabilistic distribution of (s_{t+1}, r_{t+1}) given (s_t, a_t) (s, r, a are state, reward, action respectively) after observing

$$X_t = \{(s_0, a_0), (s_1, a_1), \cdots, (s_{t-1}, a_{t-1})\}, Y_t = \{(s_1, r_1), (s_2, r_2), \cdots, (s_t, r_t)\}$$

with $t = 1, 2, \cdots$, is:

$$Prob(s_{t+1}, r_{t+1} | s_t, a_t, X_t, Y_t) \tag{4.1}$$

Let $Pred$ be the true prediction model given (s_t, a_t, p^M) and p^M is the hyper-parameter to specify a model M, which corresponds to a generalized transition matrix in an RL setting.

$$= \int_{p^M} Prob(p^M | X_t, Y_t) Pred(s_{t+1}, r_{t+1} | s_t, a_t, p^M) dp^M \tag{4.2}$$

Define a random variable θ_t which is statistically sufficient of (X_t, Y_t) to predict p^M. In other words, the following equation is satisfied: $Prob(p^M | X_t, Y_t) = Prob(p^M | \theta_t)$

$$= \int_{p^M} Prob(p^M | \theta_t) Pred(s_{t+1}, r_{t+1} | s_t, a_t, p^M) dp^M \tag{4.3}$$

Suppose $Prob(p^M|\theta_t)$ can be computed by the same probability density function $f(p^M;^{\text{i}}\theta_t)$ parameterized by θ_t.

$$= \int_{p^M} f(p^M;\theta_t) Pred(s_{t+1}, r_{t+1}|s_t, a_t, p^M) dp^M \tag{4.4}$$

In general θ_t is a function of (X_t, Y_t) with:

$$\theta = \Theta(X_t, Y_t) \tag{4.5}$$

In the context of reinforcement learning, $Pred(s_{s+1}, r_{t+1}|s_t, a_t, p^M)$ is usually referred to as the transition function and denoted $p(s_{s+1}, r_{t+1}|s_t, a_t; p^M)$, where p^M is the parameter to specify the environment dynamics.

$$\text{Equation (4.4)} = \int_{p^M} f(p^M;\theta_t) p(s_{t+1}, r_{t+1}|s_t, a_t; p^M) dp^M \tag{4.6}$$

θ_t is the belief about the environment dynamics or the hyper-parameter describing the probability distribution of parameters of transition functions, e.g., θ_t can be the means and the co-variance matrices of Gaussian distributions of parameters in p^M which specifies a model. Θ_t is a function of all the past state-action-state-reward tuples $(s_t, a_t, s_{t+1}, r_{t+1})$ and can be trivially set to memorize all the past tuples. In general, an ideal $\Theta^*(X_t, Y_t)$ function is one that makes θ_t statistically sufficient of X_t, Y_t, e.g., trivially memorizing all the past observations.

Bayesian inference can be seen as an incremental formulation of equation (4.4) with θ_0 representing *a priori* knowledge about the model. θ_t corresponds to the Bayesian belief (Prior/Posterior) in Bayesian statistics.

Assume θ_t can be updated from θ_{t-1} by a function g with

$$\theta_t = g(\theta_{t-1}, s_{t-1}, a_{t-1}, s_t, r_t) \tag{4.7}$$

such that the statistical sufficiency condition still holds

$$Prob(p^M|X_t, Y_t) = Prob(p^M|\theta_t) \tag{4.8}$$

Claim: The update rule or the function g for θ_t must have the property

$$f(p^M;\theta_t) \propto f(p^M;\theta_{t-1}) p(s_t, r_t|s_{t-1}, a_{t-1}; p^M), \forall p^M \tag{4.9}$$

[i] ';' is used when there is an explicit relation, namely f, between p^M and θ_t.

Proof:

Base case: Use *a priori* knowledge to initialize θ_0.
Inductive case: Assume θ_{n-1} is statistically sufficient of $)X_{t-1}, Y_{t-1})$ for predicting s_t, r_t

$$\begin{aligned} f(p^M;\theta_t) &= Prob(p^M|\theta_t) && (4.10)\\ &= Prob(p^M|X_t, Y_t) \ \text{ statistical sufficiency condition} && (4.11)\\ &= \frac{Prob(p^M, X_t, Y_t)}{Prob(X_t, Y_t)} = \frac{Prob(p^M, s_t, r_t, s_{t-1}, a_{t-1}, X_{t-1}, Y_{t-1})}{Prob(s_t, r_t, s_{t-1}, a_{t-1}, X_{t-1}, Y_{t-1})} && (4.12)\\ &= \frac{Prob(p^M|s_{t-1}, a_{t-1}, X_{t-1}, Y_{t-1})Prob(s_t, r_t|p^M, s_{t-1}, a_{t-1}, X_{t-1}, Y_{t-1})}{Prob(s_t, r_t|s_{t-1}, a_{t-1}, X_{t-1}, Y_{t-1})} && (4.13) \end{aligned}$$

Since p^M is independent of (s_{t-1}, a_{t-1})

$$Prob(p^M|s_{t-1}, a_{t-1}, X_{t-1}, Y_{t-1}) = Prob(p^M|X_{t-1}, Y_{t-1}) \quad (4.14)$$

Because θ_{t-1} is inductively assumed statistically sufficient

$$\begin{aligned} Prob(p^M|X_{t-1}, Y_{t-1}) &= f(p^M;\theta_{t-1}) && (4.15)\\ Prob(s_t, r_t|p^M, s_{t-1}, a_{t-1}, X_{t-1}, Y_{t-1}) &= Prob(s_t, r_t|p^M, s_{t-1}, a_{t-1}) && (4.16)\\ &= p(s_t, r_t|s_{t-1}, a_{t-1}; p^M) && (4.17) \end{aligned}$$

From Equation (4.6)

$$Prob(s_t, r_t|s_{t-1}, a_{t-1}, X_{t-1}, Y_{t-1}) = \int_{q^M} f(q^M;\theta_t)p(s_{t+1}, r_{t+1}|s_t, a_t; q^M)dq^M \quad (4.18)$$

Therefore,

$$\begin{aligned} f(p^M;\theta_t) &= \frac{f(p^M;\theta_{t-1})p(s_t, r_t|s_{t-1}, a_{t-1}, p^M)}{\int_{q^M} f(q^M;\theta_t)p(s_t, r_t|s_{t-1}, a_{t-1}; q^M)dq^M}, \forall p^M && (4.19)\\ &\propto f(p^M;\theta_{t-1})p(s_t, r_t|s_{t-1}, a_{t-1}; p^M), \forall p^M && (4.20) \end{aligned}$$

Note the update of θ_t can be expensive since it must ensure Equation (4.19) holds for all possible p^M. Even for problems of finite $\mathcal{S}$ and $\mathcal{A}$ sets each generalized transition

matrix p^M contains $\mathcal{O}(|\mathcal{S}|^2|\mathcal{A}|)$ parameters. Conjugate family of probability distributions is widely used to simplify the update rule for θ_t as it ensures $Prob(p^M;\theta_t)$ belongs to the same distribution family parameterized by θ_t. To predict the next state-reward, one still needs to compute the integration in Equation (4.6). Often in practice the state and reward space is too large to compute exact Bayesian inference. In fact, one of Equation (4.6) and (4.19) is mostly always computationally intractable. The question here is *Can we approximate the computation of Bayesian inference using neural networks?* This question will be answered later in this chapter.

Conjugate distribution family .

Equation 4.19 is not a trivial update as it is assumed implicitly that the probabilistic distribution of p^M belongs to conjugate distribution family. A general update rule of Equation 4.19 is:

$$f'(p^M;\theta_t) \propto f(p^M;\theta_{t-1})p(s_t,r_t|s_{t-1},a_{t-1};p^M) \tag{4.21}$$

f' and f are not necessarily the same in general. But the assumption that p^M follows a probability distribution in conjugate distribution family ensures that f' and f are the same at every time step. Example distributions in conjugate distribution family include Normal distribution, Beta distribution, Dirichlet distribution, etc. Beta distribution and Dirichlet distribution are the most popular as the update rule for θ_t takes only $\mathcal{O}(1)$ time. This gives some evidence to justify the usage of function approximation, as a neural network can be trained to approximate such function f.

For Bayesian adaptive reinforcement learning, one can parameterize the distribution of transition matrices p^M following *Dirichlet* distributions. As an example (without specifying rewards), suppose the environment has n states (denoted as $i = 1, 2, \ldots, n$), and assume the current state is at 1. Then For each action $a \in A$, there is a sequence of hyper-parameters $\{\alpha^a_{1,i}\}$. The probability distribution for $(p^M)^a_{1,i}$ is, where k is the normalization factor:

$$k\Pi_{i=1}^n((p^M)^a_{1,i})^{\alpha^a_{1,i}-1}, i \in \{1, 2, \cdots, n\}$$

The update rule for hyper-parameters of distributions in a conjugate family is incrementation For example, when the next state is $5(n > 5)$, the update rule for the hyper-parameters is as follows:

$$\alpha^a_{i,j} = \alpha^a_{i,j} + 1 \qquad \text{if } (i-1)(j-5) = 0 \tag{4.22}$$

$$\alpha^a_{i,j} \text{ remains the same} \qquad \text{if } (i-1)(j-5) \neq 0 \tag{4.23}$$

one also needs to update the normalization factor k.

4.2 Approximate Bayesian inference

The previous chapter presented three equivalent optimal learning formulations. The analysis in this section is mainly done in a Bayesian model based RL formulation with emphasis on planning. That is, to update the model or the belief of the distribution of the parameters of transition functions:

$$\theta_t = g(\theta_{t-1}, s_{t-1}, a_{t-1}, s_t, r_t) \tag{4.24}$$

such that

$$f(p^M; \theta_t) = \frac{f(p^M; \theta_{t-1}) p(s_t, r_t | s_{t-1}, a_{t-1}, p^M)}{\int_{q^M} f(q^M; \theta_t) p(s_t, r_t | s_{t-1}, a_{t-1}; q^M) dq^M}, \forall p^M \tag{4.25}$$

If the planning is done by learning from the simulated interactions generated by the learned model, the agent must be able to predict the next state-reward tuple:

$$Prob(s_{t+1}, r_{t+1} | s_t, a_t, X_t, Y_t) = \int_{p^M} f(p^M; \theta_t) p(s_{t+1}, r_{t+1} | s_t, a_t; p^M) dp^M \tag{4.26}$$

As an inspiration, UCB and Thompson sampling are two successful examples of approximate Bayesian inference for single state 'model-based' RL problems [16] and planning occurs in information space. Define **information state** to be (θ, s) where θ is **belief state** with Bayesian belief (Prior/Posterior) being a suitable candidate, and s_t is now renamed as **physical state** to differentiate from the belief state. For multi-state RL problems, planning in information space is notoriously difficult as it reduces to another Bellman equation.

$$V^*(s, \theta) = \max_a \{ \sum_{s', r} P(s', r | s, a; \theta) \left[r + \gamma V^*(s', \theta') \right] \} \tag{4.27}$$

where

$$\theta' = g(\theta, s, a, s', r) \tag{4.28}$$

Note that the transition model $P(s', r | s, a; \theta)$ is known here as θ fully describes the model dynamics perceived. However, previous section has shown the exact computation of Bayesian inference is in general computationally intractable. This thesis seeks to apply function approximation to the Bayesian inference in various approaches and pave the way for approximate optimal planning. Equation (4.27) also shows the connection between Bayesian RL and POMDP as θ_t can be viewed as the latent variable.

N.B. for RL problems with large state space, value functions are approximated by neural networks, which, denoted by **RL-Net** for the rest of the chapter, are essentially supervised prediction models trained with semi-gradient back propagation [16].

4.2.1 Bayes-Net and FR-Net

The insight to compute an approximate solution of the Bellman equation in information space (Equation 4.27) is to make use of neural networks as function approximators in an end-to-end manner so that intermediate computations such as integration in Equation (4.26) can be done implicitly by neural networks. The functions to approximate for Bayesian inference are two functions correlated by $\forall p^M$: Equation (4.26) and Equation (4.24) (which satisfies Equation (4.25)). Using multiple neural networks to approximate correlated functions inevitably introduces the entanglement between neural networks and co-training is proposed to address such problem. Based on co-training, meta-co-training for a class of RL problems with varying model dynamics is proposed for training neural networks to perform approximate Bayesian inference.

In Bayesian model based RL, we want to include the belief state θ_t as input to the value function neural network RL-Net so that the RL-Net has information state as its input and is able to do planning directly from the model belief instead of simulation interactions. Or equivalently as shown in Proof 3.3.1 once the latent variable is known in POMDP formulation, an agent can make optimal decision. The update of θ_n can be approximated by a **Bayes-Net** in Figure (4.2) at left, which is a recurrent neural network (RNN), due to its structural similarity to Bayesian inference that the Posterior becomes the Prior in the next round. Bayes-Net cannot be trained directly since θ_t as intermediate result is unknown. However, an indirect training method named co-training is proposed to train a **FR-Net** shown in Figure (4.2) at left, which consists of Bayes-Net as a part and additionally an FN-Net with a belief path between the two networks. **FN-Net** is a feed-forward neural network (FFN) that models $p(s_{t+1}, r_{t+1}|s_t, a_t; \theta_t)$ and the **belief path** connects the output (θ_t) of the Bayes-Net to be part of the input to the FN-Net. FR-Net can be further simplified to a single RNN as shown in the figure.

The FN-Net and the Bayes-Net in a FR-Net are *entangled* and cannot be trained separately. The reason lies in that while FN-Net and Bayes-Net are being trained to model Equation (4.26) and (4.24) that satisfies (4.25) respectively, they must agree on a belief path 'protocol' which intuitively corresponds to the type of probability distribution family, i.e., $f(p^M; \theta_t)$. FN-Net and Bayes-Net can be trained together in order to agree on a 'protocol' during training and this approach is named **co-training**. To co-train FR-Net on different θ-traces[ii], different dynamics models must be trained on and neural networks are capable of generalization from finitely many θ-traces to their neighbours. The approach to do co-training on many dynamics models is named **meta-co-training**. Meta-co-training is necessary as training on a single problem has no guarantee of learning anything useful as the

[ii] a θ-trace for a specific dynamics model is $\{\theta_0, \theta_1, \cdots, \theta_i, \cdots\}$ that converges to θ^*

FN-Net can over-fit the model dynamics without using any knowledge of θ. Nevertheless, evidence shows simple Bayesian inference rule can be learned on a single problem by simply adding a negative activity regularization term for the output layer (θ) of Bayes-Net to the loss function.

As is shown in Figure (4.1), a LSTM recurrent neural network: Bayes-Net is trained on a 4×4 maze problem to learn to perform Bayesian belief update. Below is the belief vector generated by the Bayes-Net after training on 1000 epochs of 1000 random transitions. The Bayes-Net learned to update the belief in a way similar to the update of hyper-parameters of a Dirichlet distribution where some hyper-parameters are incremented by 1 at each time step.

The reason for FR-Net being able to meta-co-train a Bayes-Net to predict θ_t is that to make accurate prediction on the next state-reward pair (s_{t+1}, r_{t+1}) from (s_t, a_t) under an arbitrary dynamics model, the best knowledge about the dynamics model must be known and Bayes-Net can only communicate to FN-Net such information through belief path. Thus, the states on the belief path, which represent the best knowledge of the model, can be interpreted as the Bayesian belief θ_t of the model or the latent variable l_t in a POMDP. At the stage of RL when the belief state along with physical state is trained on the RL-Net, the RL-Net has to conform to the 'protocol' $p(s_{t+1}, r_{t+1}|s_t, a_t; \theta_t)$ used on the belief path as well. This 'agreement' is one way when RL-Net is trained with frozen Bayes-Net and can be sped up by pre-training RL-Net with FR-Net. Active Bayes-Net throughout the training of RL agents can be used to 'negotiate' the 'protocol' and in addition to adapt to non-stationary dynamics model.

4.2.2 Sample from a pool of neural networks

As is discussed in chapter 2, Thompson sampling cannot scale for complex transition dynamics (multi-state) as there is no efficient way to compute the solution value functions of many policy evaluation equations each corresponding to a sampled p^M.

Instead of sampling p^M's and computing the solutions of the corresponding policy evaluation equations, I propose to sample the transition function $p(s_{t+1}, r_{t+1}|s_t, a_t; p^M)$ directly as a black box neural network corresponding to a p^M. Consequently, transitions of a sampled model are forward passes of that black box neural network and simulated transitions of different models can be used for planning. Compared to sampling parameters p^M's of transition functions, sampling transition functions directly cuts off unnecessary intermediate computations of computing the solution of policy evaluation equations by learning/planning on simulated transitions.

```
[[-2.0802739  -2.9999993   2.999999   -2.90658    -2.          2.
   2.33597     2.9999409  -2.999813   -3.          3.         -0.9990355
   2.8786588   3.         -3.         -2.7911572   1.9999926   2.9999754
  -0.9998462   3.         -2.          1.9999993   2.81       -1.9999995
  -3.          1.9999999   3.          3.          3.         -2.2409463
  -2.1632981  -1.9999999  -0.99421513  2.9999962  -3.          1.9999957
   2.9999936  -2.9999313   1.9999847   2.311587   -2.9999719  -1.6824409
   3.         -2.1406279  -1.9999989   3.         -2.          2.
  -1.9999945  -0.99992776  2.7347655   2.9583688  -2.9999993   2.5520062
  -0.28232917  1.9999948   1.9999981   2.9999998   3.          2.902945
  -2.999991   -3.          2.9465666   2.         -2.         -2.9999995
  -2.          3.         -1.8708187   3.         -1.9999998  -2.9999318
   2.          1.9999619   2.5102272   2.9999998  -3.          2.9999986
  -2.          3.         -2.1464503   3.         -3.          1.5238051
   2.6741023  -2.420789    2.         -2.         -0.97539407  3.
   1.6106923  -2.9999976  -1.9999776   2.8366857  -2.9999971   1.9784386
   1.1435639   2.8647847  -1.9999998   2.2770467 ]]
T 6
P 6
T [-1.66879421]
P [[-1.1853436]]
[[-3.0802739  -3.9999993   3.999998   -3.90658    -3.          3.
   3.33597     3.9999375  -3.9998     -4.          4.         -0.9999667
   3.8786511   4.         -4.         -3.7911572   2.999992    3.9999635
  -0.99968755  4.         -3.          2.9999993   3.8098824  -2.9999962
  -4.          2.9999998   4.          4.          4.         -3.2409463
  -3.1632981  -3.         -1.1854903   3.9999957  -3.9999998   2.9999907
   3.9999933  -3.9999056   2.9999838   3.311587   -3.9999332  -2.639924
   4.         -3.1406274  -2.999999    3.9999988  -3.          3.
  -2.9999943  -0.9999405   3.7347655   3.9583688  -3.9999971   3.5520062
   0.45259532  2.9999728   2.9999971   3.9999998   4.          3.902945
  -3.9999895  -3.9999998   3.946566    3.         -3.         -3.9999962
  -3.          3.9999998  -2.8708186   4.         -2.999999   -3.9999251
   3.          2.9999368   3.5102262   3.999999   -4.          3.999994
  -3.          4.         -2.8309755   4.         -4.          0.993299
   3.6740975  -3.420789    2.9999998  -3.         -0.9999932   4.
   1.6278249  -3.9999955  -2.9999712   3.8366854  -3.9999924   2.799927
   0.8890171   3.8647847  -2.9999995   3.2770317 ]]
```

Figure 4.1: The belief generated by the Bayes-Net for $t = 2, 3$

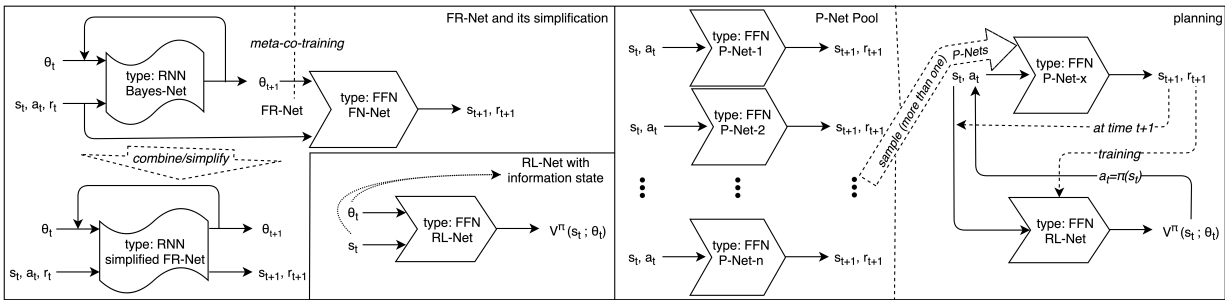


Figure 4.2: Approximate Bayesian inference.

Each neural network $p(s_{t+1}, r_{t+1}|s_t, a_t; p^M)$ corresponding to a specific p^M can be trained as a supervised prediction neural network to converge to a **P-Net**[iii] in which the weights of the neural network correspond to a p^M. For prediction, we aim to perform integration in Equation (4.26) implicitly by maintaining a pool of prediction neural networks P-Nets and generating simulated interactions by different P-Nets in the pool so that later the agent can learn to update value functions indirectly from the simulated interactions instead of the parameters of model dynamics. The P-Net pool collectively represent/encode the belief θ of the dynamics model and capture the uncertainty in the perceived knowledge about the environment. To facilitate later analysis, **model-based GPI** is introduced as an extension to generalized policy iteration(GPI)[16] which consists of policy evaluation and policy improvement. **model-based policy evaluation** becomes:

$$(\mathbf{a}.)\, V^{\pi}_{t+1}(s;\theta) = \sum_{s',r} p(s', r|s, a; \theta)\, [r + \gamma V^{\pi}_{t}(s';\theta)] \tag{4.29}$$

The value function is updated for the same policy π with the same belief θ of dynamics model. The *policy improvement* stays the same and the update for the belief of the model θ' is added.

$$(\mathbf{b}.)\, \pi'(s) = \arg\max_{a} \sum_{s',r} P(s', r|s, a; \theta)\, [r + \gamma V^{\pi}(s';\theta)] \tag{4.30}$$

$$(\mathbf{c}.)\, \theta_t = g(\theta_{t-1}, s_{t-1}, a_{t-1}, s_t, r_t) \tag{4.31}$$

such that

$$f(p^M;\theta_t) \propto f(p^M;\theta_{t-1}) p(s_t, r_t|s_{t-1}, a_{t-1}; p^M), \forall p^M \tag{4.32}$$

[iii] with input (s_t, a_t) and output (s_{t+1}, r_{t+1})

Here is the road map to make θ implicit during the entire learning.

(**a**.**b**.) The transition function $P(s', r|s, a; \theta)$ hyper-parameterized by θ can be approximated by samples of transition functions of $p(s', r|s, a; p^M)$ where p^M follows $f(\cdot; \theta)$, i.e., a pool of P-Nets, and the policy evaluation can be done by training RL-Net on simulated interactions generated from multiple transition functions sampled from the P-Net pool. As a consequence, V^π implicitly depends on θ since the training data for policy evaluation is generated from the P-Net pool in which the distribution of p^M depends on θ.

(**c**.) θ can be updated by training the P-Net pool with transitions from the true model (the real environment). As more training data is accumulated, the accuracy of the P-Nets in the pool improves, which implicates the P-Net pool becomes more certain about the model and θ skews to the true belief.

Ha and Schmidhuber[8] has proposed to learn a world model, which can be seen as a single P-Net, and train an RL agent with simulated data generated by the world model to improve data efficiency. However, the bias within the world model from the actual dynamics model can lead to poor performance in the real environment. Mixing the simulated data (planning) and real data (online training) from the true environment reduces such bias but the problem of balancing online training and planning leads to the same exploration/exploitation dilemma. From RL to OL, the P-Net pool provides a general approach to address the uncertainty in the perceived knowledge of the dynamics model by sampling the P-Net networks. In addition, since the training data for P-Net pool is from the online interaction, P-Net networks are more focused on the predictions for the states visited more frequently by agent's executing (behaviour) policy. Therefore, P-Net networks do not have to learn a complete 'world model' but a model good enough for predicting the states frequently encountered under current behaviour policy, which additionally improves data efficiency.

Training of P-Net pool By check-pointing the weights of P-Nets, training of P-Net pool can be paralleled with that of RL-Net and computation resources can be distributed freely between the two training processes depending on the training goal. We propose to train the P-Net pool using evolutionary methods as the meta-training algorithm and the training of individual networks in the P-Net pool can also be paralleled as they are independent of each other. To improve the diversity of the population in P-Net pool, we suggest constructing P-Net networks of a variety of different architectures and train them on re-organized/perturbed data.

The P-Net pool is proposed to address the uncertainty in the deduced model during learning. Since P-Net pool is an *approximate* method to do implicit Bayesian inference,

there is uncertainty in the P-Net pool:
(δ_1) finite P-Net pool space;
(δ_2) non-convergent P-Net networks;
(δ_3) inherent uncertainty of the P-Net networks trained with limited sampled observations from the environment.

δ_1 and δ_2 can be addressed by complementing the planning with learning from real interactions and the amount of planning can be quantified by $\delta_1 + \delta_2$. The δ_3 quantifies the inevitable uncertainty of the dynamics model deduced from limited past interactions with the environment and is embodied into the imperfection of P-Nets prediction and the diversity of the P-Net pool. If an infinite P-Net pool space is available and each P-Net is trained till convergence, $\delta_1 + \delta_2$ will be zero and no learning from real data is needed. Meanwhile perfect exploration/exploitation will be addressed automatically by the δ_3 in the P-Net pool implicitly. Therefore, only δ_1 and δ_2 are related to optimal exploration/exploitation rate.

In a typical RL algorithm with ϵ-greedy directing the exploration/exploitation rate, the ϵ must be non-zero and tuned empirically to the best performance. In practice finite and preferably small P-Net pool is used and δ_1 must be set in an ad hoc approach and is inversely proportionate to the size of the P-Net pool space. δ_2 can be related to the loss/accuracy changes of the P-Nets during training. The loss function here is in canonical form such as cross-entropy-like loss function so that the relationship between the change of loss function and δ_2 can be established. Bookkeeping the accuracy/loss of each state-action pair can improve the computation of δ_2 further. Though the exploration/exploitation dilemma is not completely solved in closed form because of δ_1, a clear explanation of the source of δ_1 is given and such knowledge is transferrable to other problems as it is the size of P-Net pool that decides δ_1.

Chapter 5

Approximate optimal learning algorithms and empirical evaluation

The approximate OL algorithms are constructed based on double deep Q learning and tabular Q learning for evaluation purpose and can be easily extended to most of other RL algorithms. The problems evaluated in the paper are mainly a class of randomly generated 8×8 maze problems as described in the following. The python implementation and evaluation algorithms are in https://github.com/songhobby/Approximate-Optimal-Learning.git

Randomly generated maze problems A maze consists of 64 states on an 8×8 square. There are 4 good states (which also serve as terminal states) and 4 bad states selected uniformly out of 64 total states and a set of fix-valued rewards are given randomly to the 8 special states. Action space $\mathcal{A} = \{up, down, left, right\}$ and model dynamics is defined as follows for each state-action pair:
(1) select a main resulting direction uniformly from $\mathcal{A}$;
(2) the probability of moving to direction $d \in \mathcal{A} \cup \{still\}$[i] follows a Dirichlet distribution with concentration parameters being 1 except for the main resulting direction as decided in (1) which is set to 4.

The algorithm for generating a random maze is provided as appendix in Appendix 6.1.

[i] *still* means the agent fails to move and stays at the same state

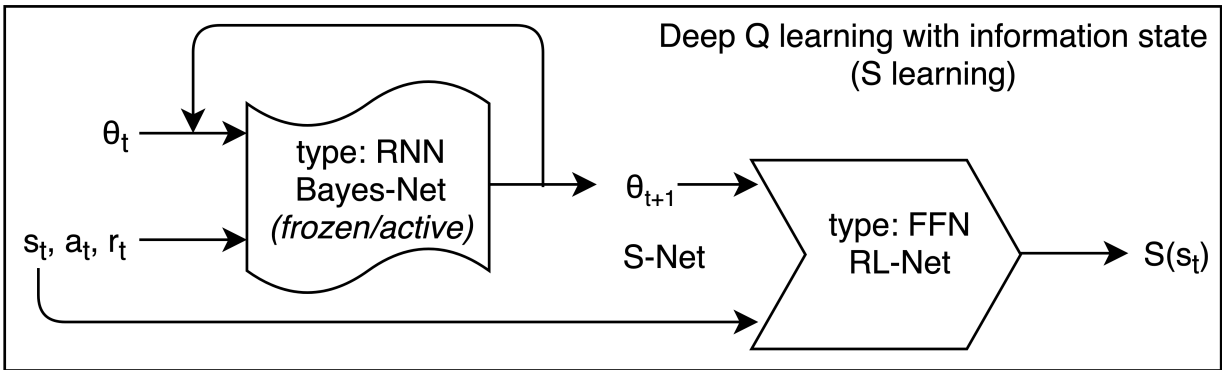
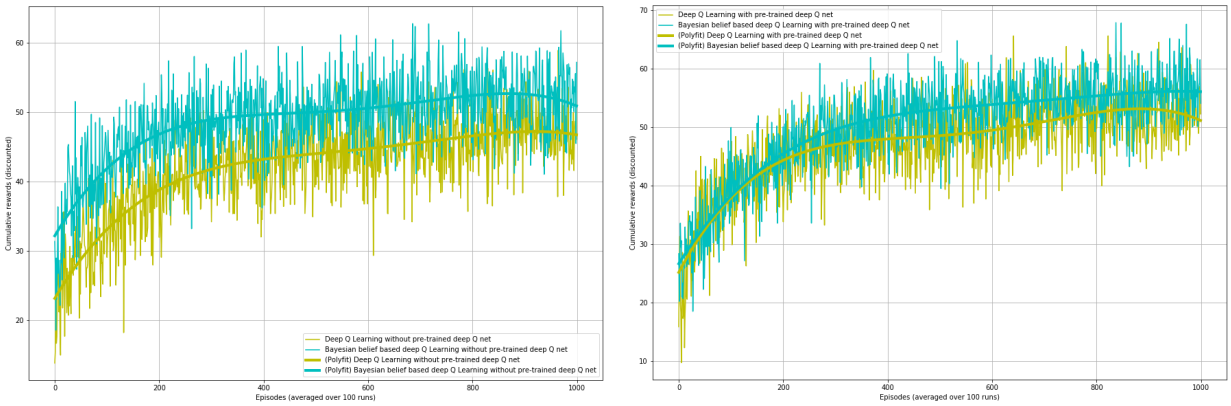


Figure 5.1: Architecture of Deep Q learning algorithm with information state (S learning)

5.1 Deep Q learning with information state (S learning)

The structure of the algorithm is pictured in Figure 5.1. S learning falls under the formulation of both POMDP and Bayesian model-based RL (directly learn from the model dynamics). On one hand θ can be seen as the latent variable being updated by the Bayes-Net with new observation. On the other hand, θ can be interpreted as the Prior from which the Bayes-Net computes the Posterior with real interaction data with the environment.

S value is defined as the value function of information state (s, θ) and Bayes-Net is pre-trained in a FR-Net. Memory based Recurrent neural network (RNN) such as Long short term memory (LSTM), Gated recurrent unit (GRU) can be built into the Bayes-Net to learn long term information about the dynamics model and FR-Net is traditional Feed forward neural network (FFN). To prevent Bayes-Net from learning nothing as the FN-Net can over-fit the dynamics model to predict next state-reward pair with no useful information of θ from belief path, Bayes-Net is meta-co-trained on many maze problems with varying parameters in the class to ensure useful information is communicated through the belief path. The method to generate a random maze problem is described at the beginning of this chapter. The Bayes-Net is then frozen during RL training on deep Q net (extends to other RL algorithms such as actor-critic RL). Active Bayes-Net can also be used for non-stationary RL problems as Bayes-Net can adapt the belief to model current dynamics by co-train the Bayes-net and RL-Net with online interactions. To further improve data efficiency, RL-Net and Bayes-Net can be trained together to agree on the distribution family protocol and the learned parameters of the RL-Net can serve as initialization during training on a specific problem. Evidence of improvement by pre-trained initialization is



(a) Without meta-co-training (b) With meta-co-training

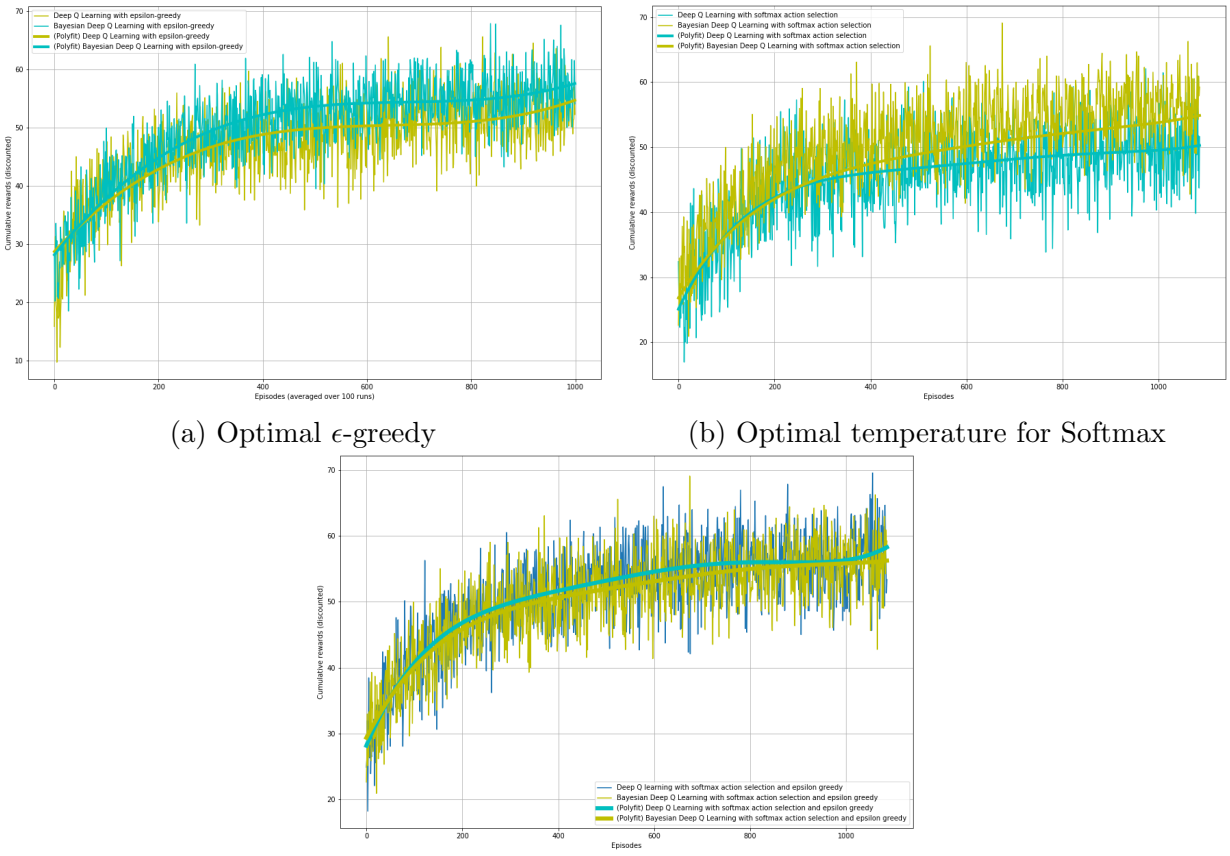
Figure 5.2: Deep Q learning with information state (S learning) averaged over 100 runs of randomly generated maze problems with/without meta-co-training

shown in Figures 5.2. Hyper-parameters ϵ and the temperature of Softmax action selection are both tuned to be optimal in the range of $[0, 0.1], [1, 10]$. Bayes-Net consists of two stacked LSTM cells, and the deep Q nets used are the same for all experiments with the belief θ being zero vector for vanilla deep Q networks for comparison purpose.

The policy learned by S learning is no worse than original RL algorithm as the belief state is additional information to the agent in a MDP formulation other than physical state. Therefore, computational resource can be freely traded for the quality of the learned policy with increased computation directed to FR-Net training.

In general using a mixture of ϵ-greedy and Softmax can achieve decent balance of exploration/eploitation as ϵ-greedy maintains a fixed amount of exploration and Softmax becomes greedy after only a few interactions. Figures (5.3) show the evidence that the use of Bayes-Net has similar performance to a vanilla double deep Q learning algorithm with a tuned mixture of ϵ-greedy and Softmax. Hyper-parameters ϵ (range $[0, 0.1]$, step size 0.01) and the temperature (range $[1, 10]$, integer) of Softmax action selection are both tuned to earliest convergence for all experiments. Bayes-Net is a single layer LSTM and is trained on about 1000 randomly generated maze problems with 4000 interactions for each problem within only 3 hours to outperform naive guess which uses the knowledge of how mazes are generated. The deep Q nets used are the same for all experiments with the belief θ zeroed for vanilla deep Q learning algorithm as comparison. Each experiment was run for about 6 hours on a single core 4.0GHz CPU and a Nvidia GTX-1060(6G memory) GPU.

S learning starts to outperform vanilla deep Q learning only after a few hundred (100 $\sim$



(a) Optimal ϵ-greedy

(b) Optimal temperature for Softmax

(c) Optimal ϵ-greedy & Softmax

Figure 5.3: Deep Q learning with information state (S learning) averaged over 100 runs of randomly generated maze problems.

300) episodes. This is an example of trading computation power for data efficiency as Bayes-Net was pre-trained. That S learning performs no better than vanilla deep Q learning at the very beginning suggests it is difficult for the Bayes-Net to infer the model dynamics without enough training data. This is an example of trading data efficiency for computation efficiency since the Bayes-Net was only trained to slightly outperform naive guess. Better Bayes-Net with more computation power towards pre-training is expected to result in fewer episodes needed for S learning to outperform vanilla deep Q learning in experiments. Intuitively, Bayesian inference under larger uncertainty is more difficult to approximate and the Bayes-Net tend to just give no information rather than wrong information about the model dynamics.

S learning may find its application in solving similar tasks repetitively such as manufacturing robotics with varied hardware to perform the same task. Data efficiency is also a significant gain in S learning that Bayes-Net can be trained on simulated models by perturbing the parameters of the original problem so that a decent Bayes-Net can be obtained and later used to train a real agent with increased data-efficiency. In general, S learning provides the flexibility to trade computation power for data efficiency.

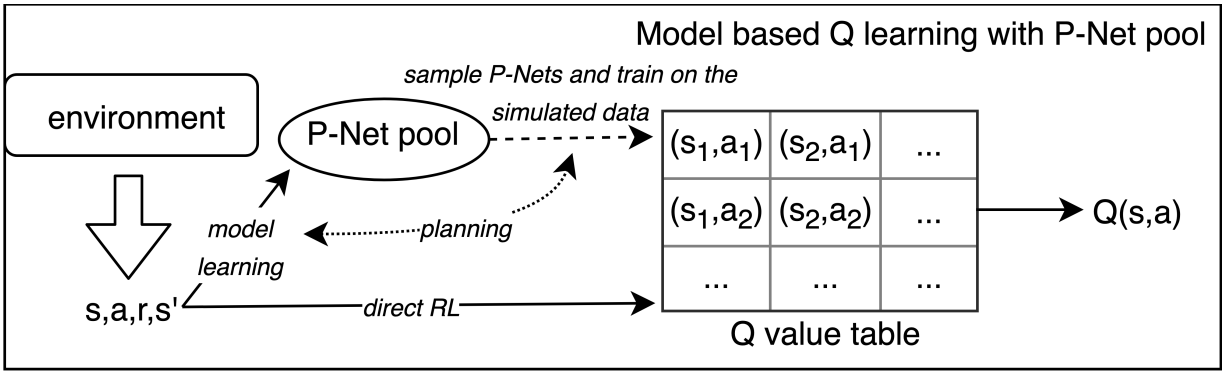


Figure 5.4: Architecture of model-based tabular Q learning by sampling neural networks (P-Net) of model dynamics

5.2 Model based reinforcement learning with P-Net pool sampling

The general architecture of the Model based Q learning with P-Net pool sampling algorithm to implicitly perform Bayesian inference is shown in Figure 5.4. The evolutionary method in use to train the P-Net pool follows principles by White[18] and obey the hypothesis 'Good genetic algorithms combine building blocks to form better solutions'. The genetic algorithm trains all the P-Nets in the pool with data from online interactions and maintains a fixed population of parent P-Nets to sample from and a fixed population of child P-Nets as candidates. The parent P-Net that has the lowest training accuracy for a few consecutive runs will be replaced by a child P-Net if the training accuracy of it is higher than the worst parent and a new born child will then be created by two existing two parents in the pool with crossover and mutation performed on the layers of the weights of neural networks.

As is shown in Figure 5.5, model-based tabular Q learning with P-Net pool outperforms the tabular Q learning by 10% of cumulative rewards in the first 200 episodes while a frequentist model in Chapter 8 by Sutton and Barto[16] showed no significant improvement in the first 200 episodes.

The design and training details of tabular Q learning with P-Net pool The P-Nets used for the evaluation of Figure 5.5 predict only the next state instead of (next state, reward) pair. The reward is hard encoded by taking the average for a (current state, action) pair to accelerate the training of the P-Net pool. The genetic algorithm to train the P-Net pool maintains a population of 10 active P-Nets of the same architecturel but different

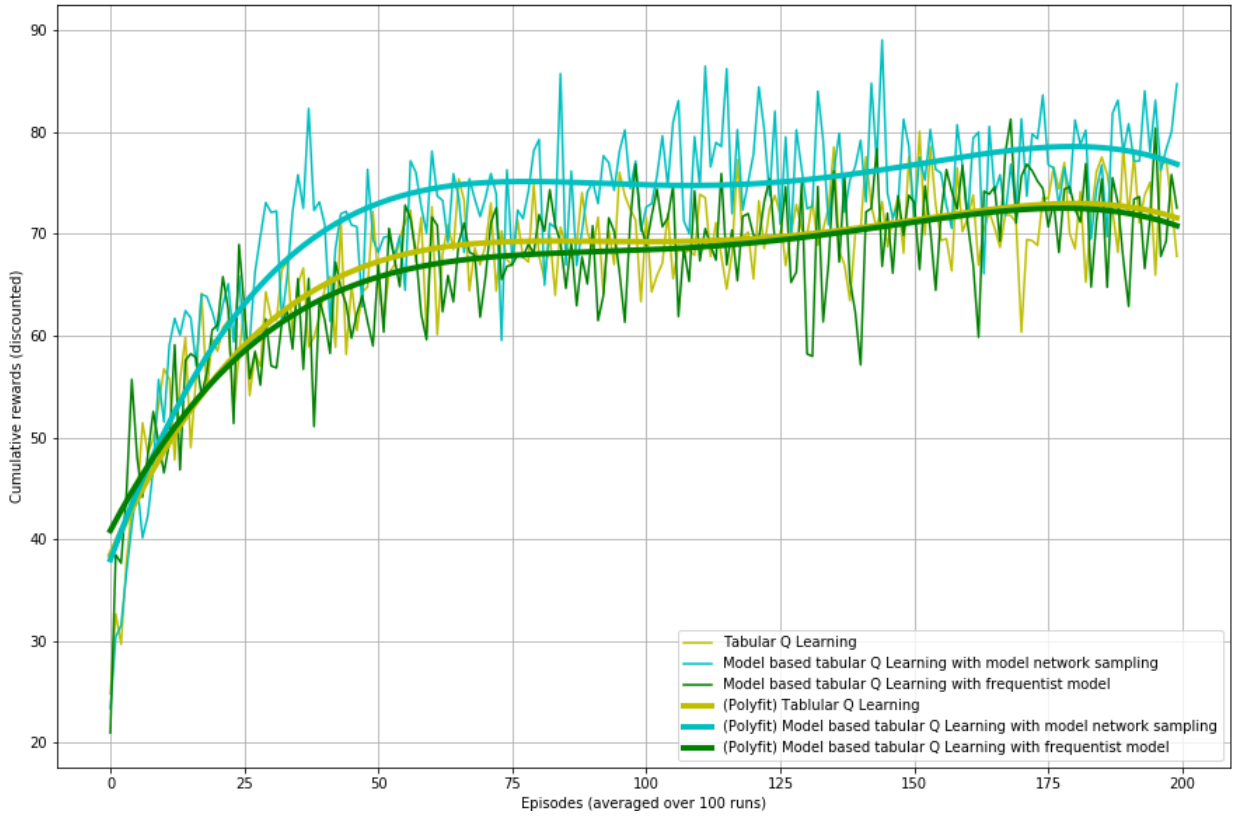


Figure 5.5: Model based tabular Q learning with different sampling techniques (averaged over 100 randomly generated maze problems.

weights and a child P-Net is reproduced by two P-Nets (parents) selected according to Boltzmann distribution. The reproduction process imitates the animal reproduction that the weights of the P-Nets resembles DNA and the weights are 'cut' into small blocks and selected randomly from either 'father' or 'mother' with the weights of an arbitrary layer of the newborn initialized randomly as 'mutation'. The python implementation used for evaluation of the evolution algorithm is provide in Appendix 6.2. The planning is performed by one sweeping update of Q functions using Bellman equation. The experiment evaluated took about 3 days to run on a single-core single-thread 4.5GHz GPU and a Nvidia 1060 (6G memory) GPU for sequential implementation given in the code repository and a parallel implementation can be done within one hour for 100 runs. N.B., we suggest paying attention to the information flow when constructing complicated neural networks to ensure all the necessary information is available to do the desired computation.

As a model-based RL, planning in the algorithm occurs in the form of interactions between the training of the P-Net pool and the improvement of value function trained with sampled P-Nets as dynamics models. The interaction is a symbiosis relationship that the training data passed to the P-Nets is affected by the improved policy, while the policy learned by any RL algorithm is improved based on the value function updated in planning with simulated data from sampled P-Nets. The P-Nets and the reinforcement learning agent influence each other throughout the entire training stage. This online interaction directs the training of P-Nets to focus on frequently visited states and therefore improves data efficiency. To effectively balance the learning from simulated data and from real online interactions, the percentage of planning can be determined by $\delta_1 + \delta_2$ as explained in the previous chapter.

Another way to understand the effectiveness of applying P-Net pool sampling is that if a pool of prediction models are able to precisely predict the next state collectively, then the perfect model must be learned and the corresponding policy must be optimal. Above is a strong argument as a statistically information sufficient model is enough to learn an optimal policy (see 4.6).

Model based reinforcement learning with P-Net pool sampling gives an approach to do approximate Bayesian inference implicitly for a single RL problem and the exploration/exploitation can therefore be resolved automatically. The fineness of approximation affects data efficiency with better approximation resulting in better data efficiency and computation resources can be traded for data efficiency by simply increasing the P-Net pool size , the expressiveness of P-Net pool or number of epochs for the training of P-Nets. The planning algorithm in model based RL with P-Net pool sampling remains unchanged for problems in continuous state-action space and most model free RL algorithms can be transformed into a model based RL with P-Net pool sampling since planning only requires simulated transitions.

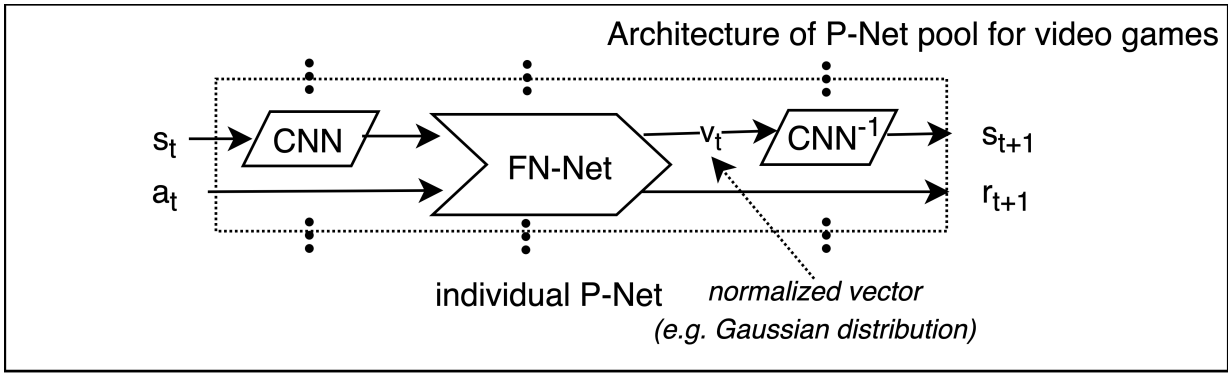


Figure 5.6: Architecture of P-Net for reinforcement learning problems based on visual input (such as video games)

P-Net pool for video games an architecture of P-Net for video games is proposed and shown in Figure 5.6.

The *state vector* v_t encodes the information for the next state in a compact form and the distribution of the state vector is normalized to follow a fixed probability distribution. Sampling v's from the fixed probability distribution corresponds to sampling states from the distribution of recently visited states under current policy. Therefore, trajectory sampling[16] can be paralleled by sampling v's independently in parallel and reverse the CNN to generate simulated transitions for planning in model based RL with P-Net pool sampling. In comparison, the conventional trajectory sampling is done by sampling states sequentially resulting from an initial state. This approach in general cannot take the advantage of distributed computing infrastructure and makes the model based RL with P-Net pool sampling even more appealing.

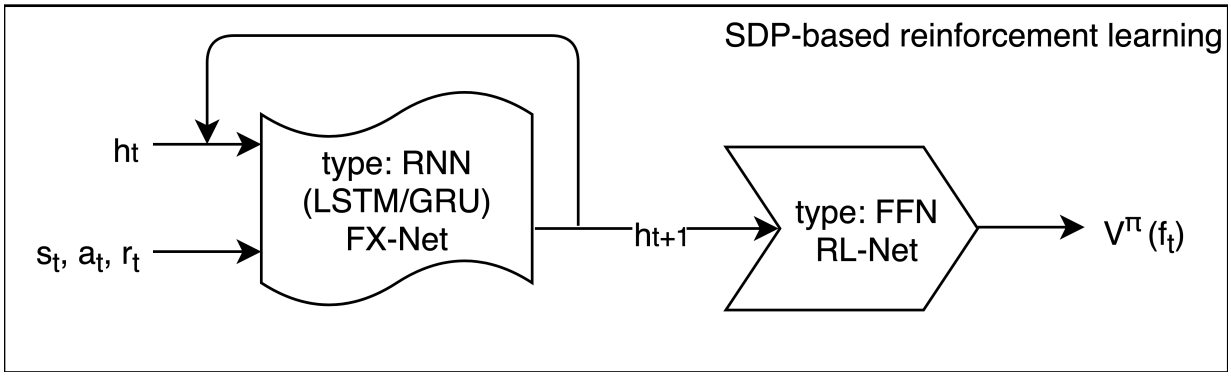


Figure 5.7: Architecture of a general stochastic decision process based reinforcement learning algorithm

5.3 Stochastic decision process (SDP) based reinforcement learning

Reinforcement learning with RNN as feature extractor has been studied to improve the RL on partially observable problems under POMDP, e.g., Hausknecht and Stone[9] trained a deep recurrent Q network to play Atari games. The success of deep recurrent Q learning is an empirical evidence of the equivalent formulations between POMDP and SDP based RL when the Deep recurrent Q learning is viewed as an SDP based RL as shown in Figure 5.7 to solve problems in POMDP.

There is other evidence to show the equivalent formulations between SDP-based RL and Bayesian model-based RL that the weakness in deep recurrent Q learning can be well explained in theory under the formulation of Bayesian model based RL. A SDP based RL can be seen as a degraded version of S learning with simplified FR-Net and without meta-co-training on a class of problems.[ii] In SDP-based RL, feature extractor (FX-Net) aims to extract h which under Bayesian model based RL corresponds to θ with the state-action pair passed into FX-Net. Extracting the state-action pair passed into FX-Net is trivial to learn but inferring the belief θ is nearly impossible. Compared with the vanilla deep Q learning, it is θ that contributes to the extra information deduced from past observations in POMDP, which corresponds to the belief of the dynamics model using Bayesian interpretation. Without meta-co-training for the FX-Net (which corresponds to

[ii] for Atari games, each class corresponds to games with varied parameters of a specific game such as Pong.

FR-Net), there is no guarantee that FX-Net can learn this extra information and as a result h either contains no extra information other than trivial information passed into the network or contains some non-consistent information that the RL-Net considers noise which it tends to ignore. To motivate FX-Net to extract the extra information about the model, memory based RNN is used to provide a path across different time steps for RL-Net to identify such extra information with the gained reward as a measure. A negative regularization term of h can be added to the loss function to encourage activation of h. Even though, identifying the extra information is as difficult as learning to perform Bayesian inference from a single problem and there is inevitable bias in learning Bayesian inference from a single problem. Consequently, there is inevitable bias in the extracted extra information that RL-Net tends to ignore at the beginning of the training and FX-Net tends to whiten this extra information which is deemed as noise by RL-Net.

We recommend using a separate neural network to extract the extra information among different states and pass the current state directly to the RL-Net to avoid the overheard of trivially extracting the information of current state and action, which are already given as input. Earlier experiment on Atari games using SDP based RL with states as direct input and a negative activation regularization term for h showed some improvement and comprehensive experiments are in progress.

Chapter 6

Summary

This thesis attempts to identify the trend in reinforcement learning research moving towards optimal learning motivated by issues as exploration/exploitation dilemma, data efficiency, partially observable MDP etc. Three equivalent optimal learning (RL learning with optimal control principles) frameworks are proposed and corresponding approximate OL algorithms are constructed. Empirical experiments were conducted to show improvements.

(1) S-learning co-trains a Bayes-Net entangled with another feed forward neural network on a class of reinforcement learning problems to approximate Bayesian inference. This Bayes-Net provides extra information (belief state) and a reinforcement learning algorithm can then be transformed to an optimal learning algorithm by augmenting the physical state with the belief state as the new input.

(2) model-based reinforcement learning with models sampled from a pool of neural networks (P-Nets) makes it possible to perform theoretical analysis. There is empirical evidence shown in chapter 5.2 that sampling from a P-Net pool can achieve a decent balance of exploration/exploitation that can otherwise only be achieved by hard tuning in an *ad hoc* approach.

(3) a more direct optimal learning algorithm is to train a reinforcement learning based on a stochastic decision process instead of a Markov decision process. However, this approach does not work well in practice since it is hard for a RL agent to identify what information should be 'memorized' and considered useful, as is explained on the weakness of recurrent deep Q learning algorithm in the last chapter.

References

[1] Richard Bellman. The theory of dynamic programming. *Bull. Amer. Math. Soc.*, 60(6):503–515, 11 1954. URL https://projecteuclid.org:443/euclid.bams/1183519147.

[2] John D Cook. Exact calculation of beta inequalities. 2005.

[3] MP Deisenroth, CE Rasmussen, and D Fox. Learning to control a low-cost manipulator using data-efficient reinforcement learning. In *Robotics: Science and Systems*, volume 7, pages 57–64, 2012.

[4] Michael O'Gordon Duff. *Optimal Learning: Computational procedures for Bayes-adaptive Markov decision processes*. PhD thesis, University of Massachusetts at Amherst, 2002.

[5] A. A. Feldbaum. Dual control theory.ii. *Avtomat. i Telemekh.*, 21, 1960. URL http://mi.mathnet.ru/at12665.

[6] Mohammad Ghavamzadeh, Shie Mannor, Joelle Pineau, and Aviv Tamar. Bayesian reinforcement learning: A survey. *CoRR*, abs/1609.04436, 2016. URL http://arxiv.org/abs/1609.04436.

[7] John C Gittins. Bandit processes and dynamic allocation indices. *Journal of the Royal Statistical Society: Series B (Methodological)*, 41(2):148–164, 1979.

[8] David Ha and Jürgen Schmidhuber. World models. *arXiv preprint arXiv:1803.10122*, 2018.

[9] Matthew Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable mdps. In *2015 AAAI Fall Symposium Series*, 2015.

[10] S. M. LaValle. *Planning Algorithms*. Cambridge University Press, Cambridge, U.K., 2006. Available at http://planning.cs.uiuc.edu/.

[11] Pascal Poupart. *Partially Observable Markov Decision Processes*, pages 754–760. Springer US, Boston, MA, 2010. ISBN 978-0-387-30164-8. doi: 10.1007/978-0-387-30164-8_629. URL https://doi.org/10.1007/978-0-387-30164-8_629.

[12] Pascal Poupart. Reinforcement learning, 2018. URL https://cs.uwaterloo.ca/~ppoupart/teaching/cs885-spring18.

[13] Herbert Robbins. Some aspects of the sequential design of experiments. *Bulletin of the American Mathematical Society*, 58(5):527–535, 1952.

[14] Malcolm Strens. A bayesian framework for reinforcement learning. In *ICML*, volume 2000, pages 943–950, 2000.

[15] Richard S Sutton. Dyna, an integrated architecture for learning, planning, and reacting. *ACM Sigart Bulletin*, 2(4):160–163, 1991.

[16] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[17] A. Wald. *Sequential Analysis*. Wiley series in probability and mathematical statistics. Probability and mathematical statistics. J. Wiley & Sons, Incorporated, 1947. URL https://books.google.ca/books?id=0nREAAAAIAAJ.

[18] David White. An overview of schema theory. *arXiv preprint arXiv:1401.2651*, 2014.

[19] Wikipedia contributors. Wikipedia, the free encyclopedia, 2018. URL https://en.wikipedia.org.

APPENDICES

6.1 Algorithm to generate a random 8×8 maze

```
''' Construct a simple maze MDP

  Grid world layout:

  ---------------------------------------
  |  0 |  1 |  2 |  3 |  4 |  5 |  6 |  7 |
  ---------------------------------------
  |  8 |  9 | 10 | 11 | 12 | 13 | 14 | 15 |
  ---------------------------------------
  ...
  ---------------------------------------
  | 56 | 57 | 58 | 59 | 60 | 61 | 62 | 63 |

  4 Goal states
  4 Bad states

  Game ends when the any goal state is reached

  4 actions (up-0, down-1, left-2, right-3).'''

# Transition function: |A| x |S| x |S'| array
import numpy as np
MAZE_SIDE_LENGTH=8
FAVOR_FACTOR=4
ACTION_SIZE=4
```

```
UP=0
DOWN=1
LEFT=2
RIGHT=3

def multi_nomial_prob_generator(action):
    alpha=[1]*(ACTION_SIZE+1)
    alpha[np.random.randint(ACTION_SIZE)]=FAVOR_FACTOR
    return np.random.dirichlet(alpha)

def look_around(state):
    return [state-MAZE_SIDE_LENGTH,state+MAZE_SIDE_LENGTH,
            state-1,state+1,state]

def naive_pred(state,action):
    if(action==UP):
        s_n = state - 8
    elif(action==DOWN):
        s_n = state + 8
    elif(action==LEFT):
        s_n = state - 1
    elif(action==RIGHT):
        s_n = state + 1
    else:
        raise Exception('Unknown Action')
    if(s_n < 0 or s_n > 63):
        s_n = state
    elif(state % 8 == 7 and s_n % 8 == 1):
        s_n = state
    elif(state % 8 == 1 and s_n % 8 == 7):
        s_n = state
    return s_n

def maze_generator():
    T = np.zeros([ACTION_SIZE,MAZE_SIDE_LENGTH**2,MAZE_SIDE_LENGTH**2])

    # 0
    for action_i in range(ACTION_SIZE):
```

```
    prob=multi_nomial_prob_generator(action_i)
    T[action_i][0][1]=prob[RIGHT]
    T[action_i][0][MAZE_SIDE_LENGTH]=prob[DOWN]
    T[action_i][0][0]=1-prob[RIGHT]-prob[DOWN]
# 7
for action_i in range(ACTION_SIZE):
    prob=multi_nomial_prob_generator(action_i)
    T[action_i][MAZE_SIDE_LENGTH-1][MAZE_SIDE_LENGTH-2]=prob[LEFT]
    T[action_i][MAZE_SIDE_LENGTH-1][2*MAZE_SIDE_LENGTH-1]=prob[DOWN]
    T[action_i][MAZE_SIDE_LENGTH-1][MAZE_SIDE_LENGTH-1]=\
    1-prob[LEFT]-prob[DOWN]

# 56
for action_i in range(ACTION_SIZE):
    prob=multi_nomial_prob_generator(action_i)
    T[action_i][MAZE_SIDE_LENGTH**2-MAZE_SIDE_LENGTH]\
    [MAZE_SIDE_LENGTH**2-2*MAZE_SIDE_LENGTH]=prob[UP]
    T[action_i][MAZE_SIDE_LENGTH**2-MAZE_SIDE_LENGTH]\
    [MAZE_SIDE_LENGTH**2-MAZE_SIDE_LENGTH+1]=prob[RIGHT]
    T[action_i][MAZE_SIDE_LENGTH**2-MAZE_SIDE_LENGTH]\
    [MAZE_SIDE_LENGTH**2-MAZE_SIDE_LENGTH]=1-prob[UP]-prob[RIGHT]

# 63
for action_i in range(ACTION_SIZE):
    prob=multi_nomial_prob_generator(action_i)
    T[action_i][MAZE_SIDE_LENGTH**2-1]\
    [MAZE_SIDE_LENGTH**2-MAZE_SIDE_LENGTH-1]=prob[UP]
    T[action_i][MAZE_SIDE_LENGTH**2-1]\
    [MAZE_SIDE_LENGTH**2-2]=prob[LEFT]
    T[action_i][MAZE_SIDE_LENGTH**2-1]\
    [MAZE_SIDE_LENGTH**2-1]=1-prob[UP]-prob[LEFT]

# 1-6
for state_i in range(1,MAZE_SIDE_LENGTH-1):
    for action_i in range(ACTION_SIZE):
        prob=multi_nomial_prob_generator(action_i)
        la=look_around(state_i)
        for la_i in range(ACTION_SIZE):
```

```
            if(la[la_i] >= 0):
                T[action_i][state_i][la[la_i]]=prob[la_i]
        T[action_i][state_i][state_i] = prob[UP]+prob[-1]

# 57-62
for state_i in range(MAZE_SIDE_LENGTH**2-MAZE_SIDE_LENGTH+1,
                    MAZE_SIDE_LENGTH**2-1):
    for action_i in range(ACTION_SIZE):
        prob=multi_nomial_prob_generator(action_i)
        la=look_around(state_i)
        for la_i in range(ACTION_SIZE):
            if(la[la_i] <= MAZE_SIDE_LENGTH**2-1):
                T[action_i][state_i][la[la_i]]=prob[la_i]
        T[action_i][state_i][state_i] = prob[DOWN]+prob[-1]

# (8,56,8)
for state_i in range(MAZE_SIDE_LENGTH,
                    MAZE_SIDE_LENGTH**2-MAZE_SIDE_LENGTH,
                    MAZE_SIDE_LENGTH):
    for action_i in range(ACTION_SIZE):
        prob=multi_nomial_prob_generator(action_i)
        la=look_around(state_i)
        for la_i in range(ACTION_SIZE):
            if(la[la_i] % MAZE_SIDE_LENGTH != (MAZE_SIDE_LENGTH-1)):
                T[action_i][state_i][la[la_i]]=prob[la_i]
        T[action_i][state_i][state_i] = prob[LEFT]+prob[-1]

# (15,63,8)
for state_i in range(2*MAZE_SIDE_LENGTH-1,
                    MAZE_SIDE_LENGTH**2-1,
                    MAZE_SIDE_LENGTH):
    for action_i in range(ACTION_SIZE):
        prob=multi_nomial_prob_generator(action_i)
        la=look_around(state_i)
        for la_i in range(ACTION_SIZE):
            if(la[la_i] % MAZE_SIDE_LENGTH != 0):
                T[action_i][state_i][la[la_i]]=prob[la_i]
        T[action_i][state_i][state_i] = prob[RIGHT]+prob[-1]
```

```
    # for the rest of the states
    for start_i in range(MAZE_SIDE_LENGTH+1,
                         MAZE_SIDE_LENGTH**2-MAZE_SIDE_LENGTH+1,
                         MAZE_SIDE_LENGTH):
        for state_i in range(start_i,start_i+MAZE_SIDE_LENGTH-2):
            for action_i in range(ACTION_SIZE):
                prob=multi_nomial_prob_generator(action_i)
                la=look_around(state_i)
                for la_i in range(ACTION_SIZE):
                    T[action_i][state_i][la[la_i]]=prob[la_i]
                T[action_i][state_i][state_i] = prob[-1]
    # Reward function: |A| x |S| array
    R = -1 * np.ones([ACTION_SIZE,MAZE_SIDE_LENGTH**2]);

    spec=np.random.permutation(np.arange(1,MAZE_SIDE_LENGTH**2))[:8]
    # set rewards
    # goal states
    R[:,spec[0]] = 100
    R[:,spec[1]] = 100
    R[:,spec[2]] = 200
    R[:,spec[3]] = 200
    E = spec[0:4]
    # bad states
    R[:,spec[4]] = -10
    R[:,spec[5]] = -10
    R[:,spec[6]] = -20
    R[:,spec[7]] = -20
    return [T,R,E]
```

6.2 Evolutionary algorithm used to train the P-Net pool

```
class Universe(object):
    # World = animal,environment,theories
```

```
    def __init__(self,parallel,life,nStates,nActions,
                 forbidden_fruit=10,adult=100,blank_paper_ad=2,
                 Temperature1=2,Temperature2=1,granularity=4):
        self.parallel = parallel
        self.life = life
        self.nStates = nStates
        self.nActions = nActions
        self.time = forbidden_fruit
        self.adult = adult
        self.blank_paper_ad = blank_paper_ad
        self.Temperature1=Temperature1
        self.Temperature2=Temperature2
        self.granularity = granularity

    def bigBang(self,rnnType,num_layers,fsi,fai,fh):
        self.worlds={}
        self.children={}
        self.prototype=BayesHypo(rnnType,num_layers,fsi,fai,fh).to(device)
        for i in range(self.parallel):
            model = BayesHypo(rnnType,num_layers,fsi,fai,fh).to(device)
            model.optimizer=model.optimizer(model.parameters())
            self.worlds[model]=-1
        for i in range(self.life):
            model = BayesHypo(rnnType,num_layers,fsi,fai,fh).to(device)
            model.optimizer=model.optimizer(model.parameters())
            self.children[model]=-1

    def world(self,Temp=1):
        lookup = self.worlds.keys()
        losses = -np.array(list(self.worlds.values()))
        prob = np.exp((losses - max(losses))/Temp)
        prob = prob / np.sum(prob)
        ret = np.random.choice(list(lookup),p=prob)
#        if(np.random.rand(1) < 0.0001):
#            print("world {} loss {}".format(id(self.worlds[ret]),self.worlds[ret]))
        return ret
```

```
    def develop(self,memory,batch_size,stablizer=0.1):

        def drama_life(model,memory):
            transitions = memory.sample(batch_size)
            batch = TransitionC(*zip(*transitions))
            state_batch,action_batch,\
            next_s_batch,reward_batch,done_batch=\
            [torch.cat(x).to(device) for x in batch]
            state_p,reward_p,done_p=model(one_hot(state_batch,self.nStates),
                                          one_hot(action_batch,self.nActions))
            loss = nn.CrossEntropyLoss()(state_p,next_s_batch)
#            +\
#                   nn.MSELoss()(reward_p,reward_batch)+\
#                   nn.BCEWithLogitsLoss()(done_p.squeeze(-1),done_batch.float())
            model.optimizer.zero_grad()
            loss.backward()
            model.optimizer.step()
            return loss.detach().item()

        self.time-=1
        weed_out =[None,-np.inf]
        join_in =[None,np.inf]
        for model,loss in self.worlds.items():
            actual_loss = drama_life(model,memory)
            if(self.worlds[model] == -1):
                self.worlds[model]=actual_loss
            else:
                self.worlds[model]=loss+stablizer*(actual_loss-loss)
            if(self.worlds[model] > weed_out[1]):
                weed_out = [model,loss]
        for i in range(self.blank_paper_ad):
            for model,loss in self.children.items():
                actual_loss = drama_life(model,memory)
                if(self.children[model] == -1):
                    self.children[model]=actual_loss
                else:
                    self.children[model]=loss+stablizer*(actual_loss-loss)
                if(i == self.blank_paper_ad-1 and
```

```
                self.children[model] < join_in[1]):
                 join_in = [model,loss]
        if(self.time < 0 and weed_out[1] >= join_in[1]):
            self.time = self.adult
            del self.worlds[weed_out[0]]
            self.worlds[join_in[0]]=join_in[1]
            del self.children[join_in[0]]
            print("\tweed_out {} {}".format(id(weed_out[0]),weed_out[1]))
            print("\tjoin_in {} {}".format(id(join_in[0]),join_in[1]))
            self.reproduce(self.Temperature1)
        elif(self.time < 0):
            print("\tweed_out {} {}".format(id(weed_out[0]),weed_out[1]))
            print("\tjoin_in {} {}".format(id(join_in[0]),join_in[1]))
            self.nirvana(self.Temperature2)
            self.time = self.adult
        return weed_out[1]

    def nirvana(self,Temperature):
        print("\tExtinction")
        for model in list(self.children.keys()):
            del self.children[model]
            del model
        for i in range(self.life):
            self.reproduce(self.Temperature2)
        print("\tReborn")


    def reproduce(self,Temperature):
        print("\tReproduce")
        lookup=list(self.worlds.keys())
        prob=np.array(list(self.worlds.values()))/float(Temperature)
        prob=np.exp(prob - max(prob))
        prob=prob / np.sum(prob)
        parents=np.random.choice(lookup,2,replace=False,p=prob)
        print("\tParents {} {}".format(id(parents[0]),id(parents[1])))
        new_born = BayesHypo(parents[0].rnnType,
                             parents[0].num_layers,
                             parents[0].fsi,
```

```
                    parents[0].fai,
                    parents[0].fh).to(device)
new_born.optimizer=new_born.optimizer(new_born.parameters())
cut = np.random.choice(len(list(self.prototype.parameters())),
                       size=random.randint(1,self.granularity),
                       replace=False)
print("\tDNA dissection {}".format(cut))
father,mother = np.random.choice(parents,2,replace=False)
DNA = OrderedDict()
dominant = father
mutation = np.random.randint(len(list(self.prototype.parameters())))
print("\tMutation {}".format(mutation))
for i,key in enumerate(father.state_dict().keys()):
    if(i == mutation):
        DNA[key] = self.prototype.state_dict()[key].clone()
    else:
        DNA[key] = dominant.state_dict()[key].clone()
    if(i in cut):
        if(dominant is father):
            dominant = mother
        else:
            dominant = father

new_born.load_state_dict(DNA)
self.children[new_born] = -1
print("\tEnd of Reproduction")
```