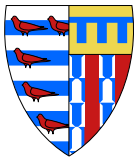
UNIVERSITY OF
CAMBRIDGE

Reinforcement Learning for Automation

Abhishek Shenoy



Pembroke College

Masters project report supervised by Dr Sumeetpal Singh

Abstract

In the real world, we encounter various stochastic decision processes which we can attempt to learn by reinforcement however the associated cost of exploration and the inability to perform exploitation in the short-term means that we require fast convergence to an adequately optimal solution.

If we now introduce the condition of partial observability to a relatively simple MDP, the optimal solution is intractable. Hence instead by considering a real-world style MDP that is stationary, we are able to compare the tractable optimal solution with the sub-optimal solutions provided by our novel methods to solve POMDPs. This is a very important problem as the applications of solving POMDPs are widely used in business, finance and engineering to solve sequential decision-making problems under imperfect observations.

We consider the Jack's Car Rental (JCR) problem defined in Sutton and Barto which is a Markov decision process (MDP). The objective is to manage the number of cars at two locations for which the demand and return of cars are Poisson distributed but different at both locations. There is an associated action cost to moving cars between the two locations and a reward for the number of cars rented out which need to be taken into account. During development we consider a model-free approach such as Q-learning which would be more useful in more complex scenarios however we find that this is not very effective due to the stochastic nature of the problem. Hence a model-based approach is investigated instead and we find that this has greater resemblance to that of real-world scenarios where an iterative cycle of model development and evaluation forms the basis of finding solutions to such scenarios. Our method of evaluation relies on uncertainty in the environment parameters from collected observations. This can be used to develop intermediate policies that are updated iteratively given more information.

Focusing on the specific aspect of utilising uncertainty, Strens outlines a Bayesian framework for reinforcement learning namely Bayesian dynamic programming (Bayesian DP). Bayesian DP maintains a Bayesian posterior over models and periodically draws a sample from this distribution. It then acts optimally with respect to this sampled model. It was shown that this method had a considerably better performance than various Q-learning approaches and Bayes VPI+MIX although only slightly better than heuristic dynamic programming on simple MDP problems.

Developing on this, we construct model parameter hypotheses and use a model-based approach using value iteration to generate multiple candidate policies which we use to test the strategies defined by Strens. We compare two main approaches specifically using the mean value matrix (VIBU-M1) and Thompson sampling (VIBU-M2) to attain a sufficiently good policy for the JCR which could theoretically be applied to POMDPs.

With the introduction of neural Q-learning, developing solutions to complex problems has become significantly easier however when considering a Bayesian approach using neural networks, we require some method of modelling uncertainty. Gal and Ghahramani show that Monte-Carlo dropout can be used as a Bayesian approximation by generating multiple Monte-Carlo outputs

from a neural network for a given input by applying dropout at both training and prediction time.

As an alternative method, we develop a model-free neural network that uses Monte-Carlo dropout (MCDQN) to compare a neural approach with the model-based value iteration approach (VIBU) both using Bayesian updates to learn the optimal policy. We compare the mean Q-vector (MCDQN-M1) and Thompson sampling (MCDQN-M2) approach for MCDQN and compare this with VIBU. We further evaluate the differences between VIBU and MCDQN as well as separately considering M1 and M2. Furthermore, increasing the number of generated candidate policies is investigated by generating more model parameters for VIBU and more Q-vectors for MCDQN. Our findings are then evaluated for the application of POMDPs with the challenging objective of obtaining feasible policies with sufficiently good performance.

Contents

Chapter 1

Introduction

There are significant problems in the real-world in which actions have to be performed to achieve optimal rewards using models of the true problem with applications ranging from science and engineering to business and finance. This is a complex problem in reinforcement learning where the only feedback received is some reward signal for a performed action in a specific state. Generally problem states in the real-world tend to be noisy or dependent on latent variables increasing the difficulty of finding a solution. Hence an iterative process is often employed to construct suitable model dynamics reflective of data obtained from real-world experience.

A simplified model problem would be to consider a Markov decision process (MDP) and hence in the noisy case, we would need to consider partially-observable MDPs (POMDPs). For MDPs, we can find optimal policies given a defined model however the parameters of the model are often unknown and need to be determined. To effectively estimate the model is a tough challenge as trial-error is expensive in the real-world. Alternatively we may consider attempting to directly find the best policy by experience. If successful, this would be extremely effective since a distribution over models can be reduced to a distribution over actions reducing computational cost and avoiding the need for a static model assumption.

In this study, we aim to develop several such strategies and explore their application for solving a toy problem. The objective is to analyse the extent of which the algorithms could apply in the real-world by evaluating their attributes and testing their performance. Being able to obtain sufficient models or policies for POMDPs is significant as solving them has major applications for effective decision-making.

Chapter 2

Background

We provide a brief overview of the details relevant to our implementations and investigations specifically introducing reinforcement learning for Markov decision processes, model-based Bayesian learning and contemporary model-free Q-learning using neural networks.

2.1 Markov Models

There are many Markov models including Markov chains (MCs), hidden Markov models (HMMs) and Markov decision processes (MDPs) as shown in 2.1. We are specifically concerned about MDPs in reinforcement learning specifically for difficult control theory problems hence allowing for controllable transitions.

Markov Models		Controllable Transitions	
		No	Yes
Observable States	Yes	MC	MDP
	No	HMM	POMDP

Table 2.1: Markov Models

2.1.1 Partial Observability

Partially observable Markov decision processes (POMDPs) are equivalent to controllable hidden Markov models. Stochastic transitions can be represented by random latent variables that vary depending on the current state and hence the chosen action.

In MDPs, we assume that the true state is always observable and the next state is dependent only on the current state and the chosen action. This assumption allows the possibility of learning an optimal reactive policy. However with the presence of latent variables in POMDPs, the observations do not obey the Markov property and hence a reactive policy found by a MDP will not be optimal.

For POMDPs, the models are characterised by the conventional transition model $P(s_{t+1}|s_t, a_t)$ and the reward function $R(s_t, a_t)$ as for MDPs as well as an additional observation dynamics model $O(o_{t+1}|s_{t+1}, a_t)$.

We define the experience/history vector $h_t = \{o_0, a_0, o_1, ..., a_{t-1}, o_t\}$ containing all past observations and performed actions.

Whereas solving MDPs is P-complete, solving POMDPs is NP-hard. To solve POMDPs there are two main possibilities, which transform a POMDP to an equivalent, but computationally

demanding MDP:

1. Information State MDP
2. Belief State MDP

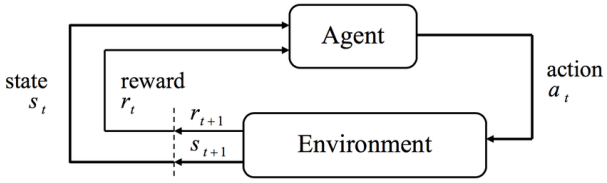
Information State Since the history vector h_t obeys the Markov property, we only need to find $P(h_{t+1}|h_t, a_t) = P(o_{t+1}|h_t, a_t)$ hence obtaining an information state MDP.

Belief State In this approach, instead of remembering the whole history which is usually infeasible, a belief function is used instead $b_t(s_t) = p(s_t|o_{1:t}, a_t)$. Using this belief state, it is possible to derive a reward and transition function. However in this case, we will have an MDP with continuous states and a method will be needed to update the belief given a new observation and action.

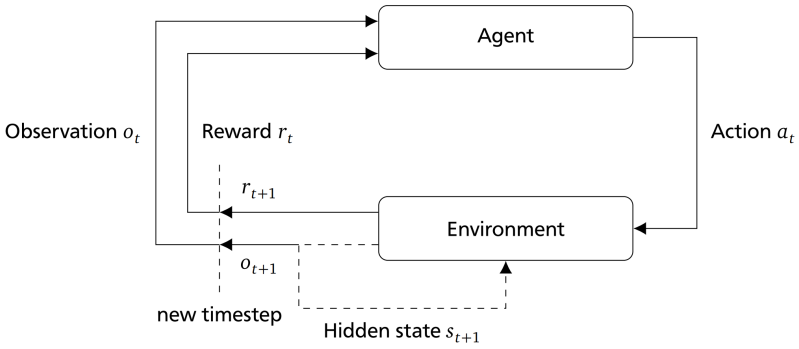
2.2 Reinforcement Learning

2.2.1 Agent-Environment Interface

In RL, we encounter a control feedback loop specifically involving an agent and the environment where the agent tries to maximise the reward given by the environment by performing actions. Figure 2.1 shows the feedback loops for both MDPs and POMDPs.



(a) MDP RL Loop



(b) POMDP RL Loop

Figure 2.1: Environment interface RL loops for MDPs and POMDPs [17]

Constructing successful POMDP RL algorithms is difficult at the very least and intractable at best since planning in POMDPs is undecidable. Hence our focus is on the exploration of algorithms for MDPs that could theoretically be applied to POMDPs.

2.2.2 Model Learning

2.2.2.1 Model-Based

Model-based RL approaches explore the environment to learn the transition probabilities $T = P(s'|s,a)$ and the rewards $R(s,a,s')$. We can use the model to compute the policy using traditional RL methods. This works well when state-space is small which is not necessarily the case in real-world scenarios. However model-based approaches offer several advantages such as being able to alter the models when real-world circumstances change and evaluating the new model by simulation or real-life testing.

2.2.2.2 Model-Free

Model-free RL approaches do not learn a model but instead attempt to learn the value function (Q-values) or policy directly. This can be extremely effective when state-space is large but drawbacks can include slow convergence and practically trial-error approaches are expensive in the real-world.

2.2.3 Traditional Methods

2.2.3.1 Q-Learning

Upon receiving reward $R(s_t, a_t)$ at the transition from a state-action pair (s_t, a_t) to the next state s_{t+1}, the Q-function is updated according to:

$$\begin{aligned} Q_k(s_t,a_t) &= Q_{k-1}(s_t,a_t) + \alpha \overbrace{\Big(R(s_t,a_t) + \gamma \max_a Q_{k-1}(s_{t+1},a) - Q_{k-1}(s_t,a_t)\Big)}^{\text{Temporal Difference}} \\ &= (1-\alpha)Q_{k-1}(s_t,a_t) + \alpha\Big(R(s_t,a_t) + \gamma \max_a Q_{k-1}(s_{t+1},a)\Big) \end{aligned}$$

Note $R(s,a)$ is simulated using the environment meaning that this is model-free unlike $\mathcal{R}(s,a)$ which is model-based. The optimal action given the current state s is $\hat{a} = \arg\max_a Q(s,a)$.

Using Q-learning offers several advantages specifically that the model-free approach can be used for more complex problems and computing the optimal policy is easier than using the value matrix V since we do not have to backtrack to find the actions that generated the value. However Q-learning can be computationally expensive when the state-action space is large. As an alternative, deep Q-learning utilises neural networks for function approximation achieving empirical breakthroughs in human-level control applications [10].

2.2.3.2 Policy Iteration

Policy iteration is a model-based approach that uses a combination of iterative processes of policy evaluation and improvement to find the optimal policy until convergence. Figure 2.2 shows the full policy iteration algorithm for a given set of model parameters z.

function $PI(z)$

assign $V_0[S]$ zeros

assign $\pi_0[S]$ zeros, $j = 1$

while $\pi_j[S]$!= $\pi_{j-1}[S]$ **do**

- Policy Evaluation -

for iteration $k \leftarrow 1$ to K **do**

for state $s \in S$ **do**

$$V_k[s] = \sum_{s'} P^{(z)}(s'|s, \pi_{j-1}[s]) \big(\mathcal{R}^{(z)}(s, \pi_{j-1}[s], s') + \gamma V_{k-1}[s']\big)$$

if $\max_s |V_k[s] - V_{k-1}[s]| < \theta$

break

- Policy Improvement -

for state $s \in S$ **do**

$$\pi_j[s] = \arg\max_a \sum_{s'} P^{(z)}(s'|s, a) \big(\mathcal{R}^{(z)}(s, a, s') + \gamma V_{k-1}[s']\big)$$

if $\pi_j[S]$!= $\pi_{j-1}[S]$

$j+ = 1$

return π_j, V_k

Figure 2.2: Policy iteration algorithm

This is an expensive process and policy improvement uses policy convergence to confirm whether a policy is optimal. In the case of POMDPs, this becomes infeasible and since convergence is not guaranteed or expected, generating intermediate policies becomes ineffective.

2.2.3.3 Value Iteration

Value iteration is a model-based approach that updates a value matrix using the model probabilities and rewards. This is computationally more feasible than policy iteration and compliant for larger state spaces unlike Q-learning.

For a set of environment/model parameters z, value iteration can be computed as follows where $P^{(z)}(s'|s, a)$ is the model-dependent transition probabilities and $\mathcal{R}^{(z)}(s, a, s')$ is the model-dependent reward:

function $VI(z)$

assign $V_0[S]$ zeros

for iteration $k \leftarrow 1$ to K **do**

for state $s \in S$ **do**

$$V_k[s] = \max_a \sum_{s'} P^{(z)}(s'|s,a)\big(\mathcal{R}^{(z)}(s,a,s') + \gamma V_{k-1}[s']\big)$$

if $\max_s |V_k[s] - V_{k-1}[s]| < \theta$

break

for state $s \in S$ **do**

$$\pi[s] = \arg\max_a \sum_{s'} P^{(z)}(s'|s,a)\big(\mathcal{R}^{(z)}(s,a,s') + \gamma V_{k-1}[s']\big)$$

return π, V_k

Figure 2.3: Value iteration algorithm

2.2.4 Challenges

There are many challenges in RL for solving MDPs and POMDPs from a model-free perspective [5] including but not limited to the following:

- **Feedback signals:** The optimal policy must only be learned through interactions with the environment. The reward is the only learning signal that the agent receives. Other information such as the transition model of the environment is not given.
- **Temporal credit assignment problem:** In many environments it is difficult to determine which actions are good, since they could have influence on results many transitions later. Thus, the agent needs somehow to be able to deal with long-term dependencies.
- **Exploration-exploitation:** The agent needs to interact with the environment to explore the environment's reward structure. However, the agent will not know if the found solution is the best hence creating a dilemma about whether the policy should be updated by focusing on exploration with sub-optimal rewards or exploit the existing policy by being greedy.
- **Curse of dimensionality**: An exponential number of samples are necessary to cover the whole state and action space. In POMDPs, we have a much higher-dimensional discrete or continuous state space.
- **Information collection-exploitation**: POMDPs have an additional challenge between information gathering and exploitation since information needs to be gathered about the true state.

Our main objective is to address the exploration-exploitation dilemma for MDPs without necessarily having to worry about dimensionality. We will consider two feedback signals specifically immediate rewards for the model-free case and immediate transitions for the model-based case (which includes the assumed model). To assign credit, we choose an appropriate discount factor that is applicable to the chosen problem and hence circumventing the need to investigate this. However a preliminary investigation may be required to choose an appropriate value for POMDPs and other MDPs. We will not directly investigate the information collection-exploitation dilemma and instead pre-select a suitable collection period by experimentation as this is only specifically applicable to POMDPs which we do not directly investigate in this project.

2.3 Bayesian Theory

For our investigations, we specifically focus on using Bayesian reinforcement learning with the aim of addressing the common challenges that occur in RL. To achieve this, we require suitable tools for estimating the MDP models including an appropriate Bayesian conjugate and sampling technique.

2.3.1 Model Posteriors

In Bayesian RL, it is common to construct a posterior distribution over models (or equivalently model parameters) for estimating the true model. Strens uses a single model sampled from a posterior but suggests that multiple models can be combined in different ways. Asmuth et al. generates multiple models from the posterior and solves the mixed MDP of all the models called as the best-of-sampled-set strategy (BOSS). The mixed MDP is constructed by keeping the existing state-space but creating an augmented action space of KA actions where each of the K models has A actions. Wilson et al. also uses a Bayesian model-based approach for attempting to solve multi-task RL problems. Wilson et al. also samples multiple models from the posterior (using a Dirichlet prior as a conjugate for the multinomial distribution) but only the model with the highest probability is selected. Dearden et al. also uses a Bayesian model-based approach using a Dirichlet prior as a conjugate for the multinomial distribution but estimate the Q-value distributions for each of the models instead of directly computing them. Instead of re-generating the Q-value distributions or "repairing" the Q-values using prioritised sweeping [11] from one set of models to another, the Q-value distributions are updated by generalisation and smoothing (for example by fitting mixture of distributions or kernel density estimation).

A common argument for model-based approaches is that by learning a model the agent can avoid costly repetition of steps in the environment by learning from simulated steps in the model [16]. Hence by sampling a model distribution, we can trial various estimates for effective exploration and exploitation. However any sampling approach must address a few key questions specifically when to sample, how many samples and how to combine models. Nonetheless, the approach of creating model posteriors is extremely useful when uncertainty is present in the model which is the case for real-world scenarios and hence holds significant applicability to POMDPs.

2.3.2 Gamma-Poisson Conjugate

For direct evaluation of Bayesian likelihoods and posteriors, we require the use of Bayesian conjugates. A commonly used conjugate for the posterior of MDP models is the Dirichlet-multinomial conjugate [19, 4]. For our investigation, an appropriate choice is the Gamma-Poisson conjugate specifically relating to Poisson-distributed arrival rates.

If our collected observations $X_1, ..., X_N$ are iid Poisson distributed $Po(\lambda)$, then the following Bayesian identities are true for a given set of samples $\{x_1, ..., x_n\}$:

$$\begin{aligned} \text{Prior} \quad & p(\lambda) = \gamma(\lambda;\ \alpha, \beta) \\ \text{Likelihood} \quad & \mathcal{L}(\lambda) = p(\boldsymbol{x}|\lambda) = \prod_{i=1}^{N} Po(x_i;\ \lambda) = \prod_{i=1}^{N} \frac{e^{-\lambda}\lambda^{x_i}}{x_i!} \\ \text{Posterior} \quad & p(\lambda|\boldsymbol{x}) = \gamma(\lambda;\ \alpha + \sum_{i=1}^{N} x_i, \beta + N) \end{aligned}$$

These update rules will be used to update the model parameters of the estimated model for our chosen problem as more experience is gathered over time.

2.3.3 Thompson Sampling

Thompson sampling is a heuristic learning algorithm in which the posterior distribution is sampled multiple times and the value that maximises the expected reward is chosen. This directly addresses the exploration-exploitation dilemma in RL problems.

For a stochastic model, we can define the method [13] as follows:

1. Choose a random model parameter from all possible model parameters.
2. Act once according to the chosen model parameter.
3. Observe the reward with the chosen model parameter.
4. Learn from this new experience and update beliefs about the possible model parameters.

Thompson sampling is the approach used by Bayesian DP [15] and shown to be effective for Bayesian RL [18].

2.4 Neural Networks

As an extension to traditional RL methods, neural networks provide a scalable way to construct RL strategies that perform significantly well [10]. Specifically Q-networks attempt to learn the optimal sum of discounted rewards (Q-values) associated with a state-action pair in a MDP which can then be used to determine the policy. Hence it is an important consideration to investigate Bayesian RL using neural learning for MDPs as well as POMDPs with many strategies achieving great success [20, 8, 2].

2.4.1 Monte-Carlo Dropout

Monte-Carlo dropout is an analogous method for generating a Bayesian posterior. Gal and Ghahramani show that Monte-Carlo dropout can be used as a Bayesian approximation by generating multiple Monte-Carlo outputs from a neural network for a given input. The Monte-Carlo dropout layer adds stochasticity in a neural network by essentially being able to generate multiple outputs for a single given input[6].

For Monte-Carlo dropout, the dropout is applied at both training and prediction time. At prediction time, the prediction is no longer deterministic and instead dependent on which nodes are randomly kept. Therefore, given the same input datapoint, the model will predict different outputs each time. Therefore we can generate a histogram of possible outputs or equivalently a predictive distribution from which to sample.

2.4.2 Q-Network

Q-networks (also referred to as deep Q-networks (DQNs) in literature) aim to learn the Q-values for a given state and action using a neural network.

2.4.2.1 Experience

We store the agent's experience $e_t = (s_t, a_t, r_t, s_{t+1})$ at each timestep t in a dataset $\mathcal{E} = \{e_1, ..., e_P\}$ pooled over many episodes into a first-in-first-out replay memory buffer. We then sample the memory randomly to break temporal correlations obtaining a minibatch of B experience vectors, and use this to learn off-policy, as usually done with Q-networks. Sampling the memory tackles the problem of auto-correlation leading to unstable training by converting the problem to a supervised learning problem.

2.4.2.2 Experience Replay

Training occurs only during experience replay. Let $Q(s)$ be the true Q-function and $\mathcal{Q}(s;\theta)$ the Q-function approximator where θ are the parameters that characterise the function approximator which in this case are the weights of the neural network. Therefore the loss function is given by $L(\theta)$ for batch size B sampled from the experience memory [7].

$$L(\theta) = \frac{1}{B}\sum_{i=1}^{B}(Q(s_i, a_i) - \mathcal{Q}(s_i, a_i; \theta))^2$$

Since we do not have access to the true action values ($Q(s_i, a_i)$ is not known), we have to use a bootstrapped version of the last estimator accounting for the new experience and reward given[9]:

$$Q(s_t, a_t) = r_t + \gamma \max_{a_{t+1} \in \mathcal{A}} \mathcal{Q}(s_{t+1}, a_{t+1}; \theta)$$

This also shows the limitation that Q-networks are only applicable for discrete/discretised action-spaces, since we need to be able to take the max operator over all actions.

We then apply gradient descent to the parameters θ to update the weights:

$$\theta_{n+1} = \theta_n - \alpha \nabla L(\theta_n)$$

2.4.2.3 Algorithm

The full Q-network algorithm has been shown in Figure 2.4 which will be modified for the Q-network strategy in Section 3.3. In the conventional Q-network, the exploration-exploitation dilemma is attempted to be resolved using epsilon-based action-space exploration. Epsilon-based exploration picks a random action with probability ϵ and an on-policy action with probability $1 - \epsilon$ where ϵ decays over time with many existing algorithms for epsilon-decay.

$$\hat{a} = \begin{cases} \text{random } a \in \mathcal{A} & p = \epsilon \\ \arg\max_a \mathcal{Q}(s, a; \theta_n) & p = 1 - \epsilon \end{cases}$$

In Section 3.3, we will construct alternative action selection strategies to provide a better solution to the exploration-exploitation dilemma.

assign $\mathcal{E} = \{\}$ empty experience buffer

assign $\mathcal{Q} \leftarrow \theta_{n=0}$ random network weights

for episode $e \leftarrow 1$ to L **do**

$s \leftarrow$ reset env

for step $t \leftarrow 1$ to T **do**

$$\hat{a} = \begin{cases} \text{random } a \in \mathcal{A} & p = \epsilon \\ \arg\max_a \mathcal{Q}(s, a; \theta_n) & p = 1 - \epsilon \end{cases}$$

$s',\ r,\ \text{done} \leftarrow$ simulate env $\hat{a}$

$\mathcal{E} \xleftarrow{+} (s, a, r, s', \text{done})$

$s = s'$

if done **do**

break

if $|\mathcal{E}| \geq B$ **do**

$\{\ \underline{e}_j\ \}_{j=1}^{B} \sim \mathcal{E}$ Sample B experiences from $\mathcal{E}$

$$\{Q_j\}_{j=1}^{B} = \begin{cases} r_j & \text{done}_j = \text{True} \\ r_j + \gamma \max_{a'} \mathcal{Q}(s'_j, a'; \theta_n) & \text{done}_j = \text{False} \end{cases}$$

$$L(\theta) = \frac{1}{B} \sum_{i=1}^{B} (Q_i - \mathcal{Q}(s_i, a_i; \theta_n))^2$$

$$\theta_{n+1} = \theta_n - \alpha \nabla L(\theta_n)$$

Figure 2.4: Q-Network algorithm

Chapter 3

Implementations

For our investigation, we need to select a problem to investigate and construct the strategies that will be implemented to solve the problem. We specifically consider a model-based (VIBU) and a model-free (MCDQN) strategy for the Jack's Car Rental (JCR) problem. This chapter describes the implementations of the chosen problem (JCR in Section 3.1) and the strategies investigated (VIBU in Section 3.2 and MCDQN in 3.3).

3.1 Jack's Car Rental Problem (JCR)

3.1.1 Problem Choice

We explored a variety of problems such as the Frozen Lake MDP as well as a preliminary investigation of Tetris however both were found to be unsuitable for this investigation with the former problem being too simple and the latter having little application of uncertainty. Hence JCR was selected since it is a simple MDP and models a real-world scenario.

3.1.2 Problem Definition

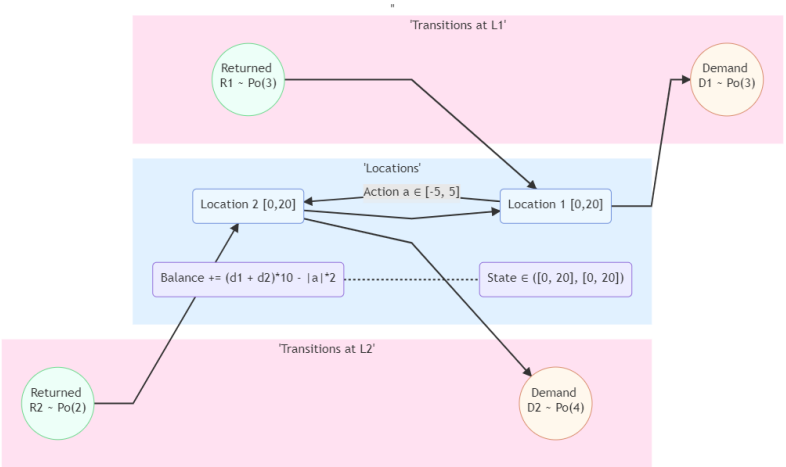


Figure 3.1: Jack's Car Rental

Jack manages two locations for a car rental company. Each day, some number of customers arrive at each location to rent cars. If Jack has a car available, he rents it out and is credited 10. If he is out of cars at that location, then the business is lost. Cars become available for renting the day after they are returned. To help ensure that cars are available where they are needed,

Jack can move them between the two locations overnight, at a cost of 2 per car moved. We assume that the number of cars requested $D^{(l)}$ and returned $R^{(l)}$ at each location l are Poisson random variables, specifically $D^{(1)} \sim Po(3), R^{(1)} \sim Po(3)$ and $D^{(2)} \sim Po(4), R^{(2)} \sim Po(2)$.

To simplify the problem slightly, we assume that there can be no more than 20 cars at each location (any additional cars are returned to the company, and thus disappear from the problem) and a maximum of five cars can be moved from one location to the other in one night. We take the discount rate to be $\gamma = 0.9$ and formulate this as a continuing finite MDP, where the time steps are days, the state is the number of cars at each location $\underline{s} = \begin{pmatrix} s_1 \\ s_2 \end{pmatrix}$ at the end of the day, and the action a is the net numbers of cars moved between the two locations overnight ($|a| \leq 5$).

Since we use a model based approach, we require the knowledge of the transition probabilities $P(\underline{s}'|\underline{s}, a)$ and the reward function $\mathcal{R}(\underline{s}, a)$ or analogously $\mathcal{R}(\underline{s}, a, \underline{s}')$:

$$\mathcal{R}(\underline{s}, a) = \sum_{\underline{s}' \in \mathcal{S}} P(\underline{s}'|\underline{s}, a)\mathcal{R}(\underline{s}, a, \underline{s}') = \sum_{\underline{s}' \in \mathcal{S}} \mathbb{E}\Big[R_{t+1}|S_t = \underline{s}, A_t = a, S_{t+1} = \underline{s}'\Big] = \sum_{\underline{s}' \in \mathcal{S}} \int_r r \cdot P(\underline{s}', r|\underline{s}, a)dr$$

For the JCR, these are directly computable given the environment parameters. The problem environment was modelled (Figure 3.1) and implemented using the Gym standard API in Python.

3.1.3 Model Derivation

This is an example of a discrete state problem, with state-space 21x21 and action-space -5 to 5 with the real actions being limited by the numbers of the cars in the locations.

There are a maximum of $M = 20$ cars allowed on each of location 1 and 2 with both locations having their own independent rent and return rates of cars.

The Poisson means are chosen to be $\lambda_d^{(1)} = 3, \lambda_r^{(1)} = 3, \lambda_d^{(2)} = 4, \lambda_r^{(2)} = 2$.

Rent earns 10 per car but moving cars between the locations costs 2 per car. The game ends if either location receives a demand it cannot fulfil completely in the same day.

3.1.3.1 Stochastic Transitions

Since the game ends if demand cannot be fulfilled and the location is limited to a maximum number of cars, both the demand and return random variables need to be clipped to calculate the transition probabilities.

The rental requests (the demand) $D_C^{(l)}$ at location l is a clipped Poisson random variable Po_C meaning the tail probability is placed at C (where C is the maximum possible that can be rented till there are 0 cars at l).

$$D_C^{(l)} \sim Po_C(\lambda_d^{(l)})$$

The returns $R_C^{(l)}$ at location l is a clipped Poisson random variable Po_C meaning the tail probability is placed at C (where C is the maximum possible that can be returned till there are M cars at l).

$$R_C^{(l)} \sim Po_C(\lambda_r^{(l)})$$

3.1.3.2 State-Action Definitions

The components of the state $\underline{s} = \begin{bmatrix} s_1 \\ s_2 \end{bmatrix}$ are the cars at each location. The integer action $a \in [-5, 5]$ is the number of cars moved from location 1 to 2.

We can define an **intermediate state** $\underline{\hat{s}} = \begin{bmatrix} \hat{s_1} \\ \hat{s_2} \end{bmatrix} = \begin{bmatrix} s_1 - a \\ s_2 + a \end{bmatrix}$. This is how the state changes when an action is taken.

After an action is chosen, there is an additional **stochastic transition** from $\underline{\hat{s}}$ to the next state $\underline{s}'$ due to the demand and return of cars:

$$d_1 \leftarrow D_{\hat{s}_1}^{(1)} \qquad r_1 \leftarrow R_{M-\hat{s_1}+d_1}^{(1)} \qquad d_2 \leftarrow D_{\hat{s}_2}^{(2)} \qquad r_2 \leftarrow R_{M-\hat{s_2}+d_2}^{(2)}$$

$$\underline{s}' = \begin{bmatrix} s_1' \\ s_2' \end{bmatrix} = \begin{bmatrix} \hat{s}_1 - d_1 + r_1 \\ \hat{s}_2 - d_2 + r_2 \end{bmatrix}$$

Note that the clipped random variables are being used to indicate that the game ends if any other values are chosen other than those within the interval of the clipping.

3.1.3.3 Transition Probabilities

$$P(\underline{s}'|\underline{s}, a) = P(\underline{s}'|\underline{\hat{s}}) = P(s_1'|\hat{s}_1) \cdot P(s_2'|\hat{s}_2)$$

We can calculate $P(s_l'|\hat{s}_l)$ for $l \in \{1, 2\}$ using the following where ρ is the reward specific to location l (used to avoid getting confused with r which is the number of returned cars):

$$P(s_l'|\hat{s}_l) = \sum_{\rho} P(s_l', \rho|\hat{s}_l) = \sum_{\substack{d \in [0, \hat{s}_l],\ r \in [0, M-\hat{s}_l+d] \\ \text{for which } s_l' = \hat{s}_l - d + r}} P(d, r|\hat{s}_l)$$

Note that d and r are also specific to location l as mentioned in the state-action definitions.

PMF Tables for d, r given $\hat{s}_l$

We precompute the **constant PMF table** for all values of $\hat{s}_l \in [0, M]$, $d \in [0, M]$ and $r \in [0, M]$ ($(M+1) \times (M+1) \times (M+1)$ matrix, one for each location l).

$$P(d, r|\hat{s}_l) = P(D_{\hat{s}_l}^{(l)} = d) \cdot P(R_{M-\hat{s}_l+d}^{(l)} = r)$$

3.1.3.4 Reward Function

In literature, this is usually defined as $R(s, a, s')$. In many cases this is a deterministic function usually dependent only on s'. In the case of JCR, this is stochastic as there are multiple transitions (random pick of d and r) that lead to s' from $\hat{s}$ and the reward is dependent on the transition and not directly on s'. Therefore we consider the expected reward for $\hat{s} \to s'$ which is

the sum over all transitions for $\hat{s} \to s'$ of the transition probability multiplied by the reward of that transition.

$$\mathcal{R}_l(\hat{s}_l, s'_l) = \mathbb{E}(\mathcal{R}_l|\hat{s}_l, s'_l) = \sum_{\rho} \rho \; P(s'_l, \rho|\hat{s}_l) = \sum_{\substack{d\in[0,\hat{s}_l],\; r\in[0,M-\hat{s}_l+d] \\ \text{for which } s'_l=\hat{s}_l-d+r}} \underbrace{(d\cdot 10)}_{\text{Reward from rent}} \cdot \; P(d, r|\hat{s}_l)$$

Note however that this only calculates the reward for location l and not the total reward. So now we calculate the total reward from the combined locations as well as the cost of the action.

$$\mathcal{R}_T(\underline{s}, a, \underline{s}') = \frac{\mathbb{E}(\mathcal{R}_1|\hat{s}_1, s'_1)}{P(s'_1|\hat{s}_1)} + \frac{\mathbb{E}(\mathcal{R}_2|\hat{s}_2, s'_2)}{P(s'_2|\hat{s}_2)} - \underbrace{|a|\cdot 2}_{\text{Cost of action}}$$

We must divide the individual reward functions $\mathbb{E}(\mathcal{R}_l|\hat{s}_l, s'_l)$ by the probability of going from $\hat{s} \to s'$ (ie. the total sum of the probabilities all transitions for $\hat{s} \to s'$ as this is not equal to 1 as there are multiple possible s'). The division normalises the reward for each location since we used $p(s', \rho|s, a)$ to calculate the expected reward so we must divide by $p(s'|s, a)$ for normalisation.

We now have both $P(\underline{s}'|\underline{s}, a)$ and $\mathcal{R}_T(\underline{s}, a, \underline{s}')$ which can be used for model-based RL.

(a) Policy Iteration 1

(b) Policy Iteration 2

(c) Policy Iteration 3

(d) Policy Iteration 4

Figure 3.2: JCR Policy Iteration

We validate that the JCR model is correctly implemented by replicating the results in Sutton and Barto [17] for the JCR obtaining the correct optimal policy for the specified parameters and showing convergence within 4 policy iterations in Figure 3.2.

3.2 Value Iteration Bayesian Update (VIBU)

We now implement a strategy defined by Strens which generates model parameters as hypotheses and updates them according to a Bayesian method. Since we have constructed the JCR model dynamics (Section 3.1.3), given any model parameters we can generate the hypothesised model and hence the corresponding optimal policy. Hence given a set of candidate model parameters, we can generate a set of candidate policies using some RL method (Section 2.2.3).

3.2.1 Candidate Policy Generation

Therefore we require a reinforcement learning method for generating these candidate policies and hence we analyse traditional RL methods to investigate which method should be used for developing our solution. Note that all the conventional methods stated in Section 2.2.3 are guaranteed to converge to the optimal solution for fully observable MDPs albeit with different convergence rates. The optimal policy that is found is only applicable for the set of model parameters that were used (for the model-based approaches). However in the case of POMDPs, this can be used to generate intermediate/candidate policies using belief vectors.

For the JCR problem, we find that Q-learning requires many iterations for a reasonable policy and does not feasibly find the optimal policy hence we opt to use a model-based approach specifically using value iteration to generate candidate policies. The agent will generate these policies by creating hypotheses about the environment parameters by attempting to characterise the demand and return means and then using value iteration to find the hypothesised policy. These policies will be generated and updated automatically as the agent learns more about the parameters.

3.2.1.1 Gamma-Poisson Conjugate

The environment parameters of JCR are Poisson random variables and hence the gamma-poisson conjugate is used for the Bayesian approach. The posterior distribution is a gamma distribution of the form $\gamma(\alpha + \sum_{i=1}^{N} x_i, \beta + N)$ where α and β are the prior parameters and $\{x_i\}_{i=1}^{N}$ are the samples collected of the random variables (Section 2.3.2). For JCR, these are the stochastic transitions that occur after a step due to the stochastic demand and return of cars (Section 3.1.3.1).

We can assume an arbitrary but appropriate prior since as $N \to \infty$, the posterior mean $\hat{\lambda} = \frac{\alpha + \sum_{i=1}^{N} x_i}{\beta + N}$ tends to the true Poisson mean λ. This is the Bayes estimate under MSE for the Gamma-Poisson conjugate. We do not specifically use the Bayes estimator for generating a policy since we could use many alternate risk functions to give different estimates such as the posterior median or the posterior mode. Instead we sample the posterior to generate parameter estimates [12].

For each of these set of parameter estimates, we compute a candidate policy using value iteration hence creating a set of candidate policies. We let the prior parameters equal $\alpha = \beta = 1$ as these are not far from the true parameter values while providing enough uncertainty to test algorithm performance.

3.2.2 Bayesian Policy Selection

Our aim is to select, combine or unify a policy from the generated candidate policies by building on a traditional reinforcement learning method with a Bayesian update such that a sufficient solution can be found for POMDPs.

Strens outlines a few approaches of selecting the optimal action. Below we develop, implement and compare 3 of these methods for the JCR to see if these can be applied to POMDPs (Section 3.2.2.1, 3.2.2.2, 3.2.2.3). The methods sample the observation posteriors N times and compute the policy for each sample using value iteration (optimal for that sample) hence generating N candidate policies. The methods then generate the overall policy using the generated candidates in different ways.

Strens refers to using the expected discounted return to determine the candidate policies. Since the value for a given state is the expected discounted return given the state ($V_\pi[s] = \mathbb{E}_\pi[G_t|S_t = s]$), we can simply use the value matrices in our methods. Similarly if we were to use the value given a state and action, we would use the Q-matrix as $Q_\pi[s, a] = \mathbb{E}_\pi[G_t|S_t = s, A_t = a]$.

Note that for a stationary process, the posterior distribution would converge very quickly (as in the case of the JCR) but sampling the posterior to generate candidates is useful in the case of a POMDP when the environment parameter means could vary with time (ie. when the process is not stationary). Sampling the posterior distribution provides uncertainty information about the parameters and hence a better estimate of the policy at that point in time.

The VIBU algorithm consists of 3 parts, namely action selection, posterior update and policy update. Action selection uses the current overall policy to choose the current action on every step. The posterior is updated on every step using the Gamma-Poisson conjugate update rule. The policy is updated every 4 episodes where the frequency of the update was chosen by experimentation. Below we present the algorithms in their full form while noting the action selection and policy update rules specific to the strategies. The performance of the various strategies are presented in Section 4.2 and 4.3.

3.2.2.1 Average Discounted Return $\mathcal{M}1$

In this strategy, a set of candidate value matrices are generated using value iteration (defined in Figure 2.3) from the posterior parameters samples. We average the value matrices across the states to construct a new value matrix V_{avg}. This value matrix is then used to determine the policy. The full algorithm is presented in Figure 3.3.

This strategy can be useful as initial outliers in value estimates will be effectively eliminated creating a more exploratory strategy for states where variance is high and a more exploitative strategy for states where variance of V is low [15].

Initialise parameters for prior Gamma distributions

$\alpha^{(R_1)}, \beta^{(R_1)} = 1, 1 \qquad \alpha^{(R_2)}, \beta^{(R_2)} = 1, 1 \qquad \alpha^{(D_1)}, \beta^{(D_1)} = 1, 1 \qquad \alpha^{(D_2)}, \beta^{(D_2)} = 1, 1$

$\pi_{\text{overall}} = \pi_{\emptyset}$ Initialise overall policy to null policy

$N = 10, L = 20, T = 200$

for episode $e \leftarrow 1$ to L **do**

  $s \leftarrow$ reset env

  for step $t \leftarrow 1$ to T **do**

    $\hat{a} = \pi_{\text{overall}}[s]$ M1 Action Selection

    $s',\ r,\ \text{done},\ [r_1, r_2, d_1, d_2] \leftarrow$ simulate env $\hat{a}$

    Update Posteriors

    $\alpha^{(R_1)} := \alpha^{(R_1)} + r_1;\ \beta^{(R_1)} := \beta^{(R_1)} + 1; \qquad \alpha^{(R_2)} := \alpha^{(R_2)} + r_2;\ \beta^{(R_2)} := \beta^{(R_2)} + 1;$

    $\alpha^{(D_1)} := \alpha^{(D_1)} + d_1;\ \beta^{(D_1)} := \beta^{(D_1)} + 1; \qquad \alpha^{(D_2)} := \alpha^{(D_2)} + d_2;\ \beta^{(D_2)} := \beta^{(D_2)} + 1;$

    $s = s'$

    if done **do**

      break

  if $e \ \% \ 4 = 0$ **do**

    Create posteriors using stored parameters

    $R_1 \sim \gamma(\alpha^{(R_1)}, \beta^{(R_1)});\ R_2 \sim \gamma(\alpha^{(R_2)}, \beta^{(R_2)})$

    $D_1 \sim \gamma(\alpha^{(D_1)}, \beta^{(D_1)});\ D_2 \sim \gamma(\alpha^{(D_2)}, \beta^{(D_2)})$

    Start - M1 Policy Update

    for candidate sample $c \leftarrow 1$ to N **do**

      Generate sample from independent posteriors

      $z_c = [r_1 \leftarrow R_1, r_2 \leftarrow R_2, d_1 \leftarrow D_1, d_2 \leftarrow D_2]$

      Generate candidate V from sample via value iteration

      $V_c \leftarrow \text{VI}(z_c)$

    Calculate average predicted discounted return

    $V_{\text{avg}} = \frac{1}{N} \sum_{c=1}^{N} V_c$

    Computing the overall policy is necessary

    $\pi_{\text{overall}} = \pi_{\emptyset}$

    for state $s \in S$ **do**

      $\pi_{\text{overall}}[s] = \arg\max_a \sum_{s'} P(s'|s,a)\big(R(s,a,s') + \gamma V_{\text{avg}}[s']\big)$

    End - M1 Policy Update

Figure 3.3: VIBU-M1 Algorithm

Computing the full policy is required in this method but this is computationally cheaper than VIBU-M2, where a policy is computed for each candidate sample z_c.

3.2.2.2 Most Common Action $\mathcal{M}2$

This strategy considers Thompson sampling for generating the policy [13]. A candidate policy is directly generated using value iteration (defined in Figure 2.3) for every set of posterior parameter samples. The overall policy is the most common action from all the policies given a specific state. The full algorithm is presented in Figure 3.4.

Since value iteration generates the policy with the greatest expected reward, sampling the posterior to generate a set of candidate policies via value iteration and then picking the most common action of all the policies is equivalent to Thompson sampling.

Initialise parameters for prior Gamma distributions

$\alpha^{(R_1)}, \beta^{(R_1)} = 1, 1 \qquad \alpha^{(R_2)}, \beta^{(R_2)} = 1, 1 \qquad \alpha^{(D_1)}, \beta^{(D_1)} = 1, 1 \qquad \alpha^{(D_2)}, \beta^{(D_2)} = 1, 1$

$\Pi_C = \{\pi_\emptyset\}$ Initialise candidate policies to null policy

$N = 10, L = 20, T = 200$

for episode $e \leftarrow 1$ to L **do**

$s \leftarrow$ reset env

for step $t \leftarrow 1$ to T **do**

$\mathcal{A}_C = \{a_c = \pi_c[s]\}_{\pi_c \in \Pi_C}$ M2 Candidate Actions

$\hat{a} = \arg\max_a \sum_{a_c \in \mathcal{A}_C} \mathbb{I}[a_c = a]$ M2 Action Selection

$s',\ r,\ \text{done},\ [r_1, r_2, d_1, d_2] \leftarrow$ simulate env $\hat{a}$

Update Posteriors

$\alpha^{(R_1)} := \alpha^{(R_1)} + r_1;\ \beta^{(R_1)} := \beta^{(R_1)} + 1; \qquad \alpha^{(R_2)} := \alpha^{(R_2)} + r_2;\ \beta^{(R_2)} := \beta^{(R_2)} + 1;$

$\alpha^{(D_1)} := \alpha^{(D_1)} + d_1;\ \beta^{(D_1)} := \beta^{(D_1)} + 1; \qquad \alpha^{(D_2)} := \alpha^{(D_2)} + d_2;\ \beta^{(D_2)} := \beta^{(D_2)} + 1;$

$s = s'$

if done **do**

break

if $e \ \%\ 4 = 0$ **do**

Create posteriors using stored parameters

$R_1 \sim \gamma(\alpha^{(R_1)}, \beta^{(R_1)});\ R_2 \sim \gamma(\alpha^{(R_2)}, \beta^{(R_2)})$

$D_1 \sim \gamma(\alpha^{(D_1)}, \beta^{(D_1)});\ D_2 \sim \gamma(\alpha^{(D_2)}, \beta^{(D_2)})$

Start - M2 Policy Update

$\Pi_C = \{\}$

for candidate sample $c \leftarrow 1$ to N **do**

Generate sample from independent posteriors

$z_c = [r_1 \leftarrow R_1, r_2 \leftarrow R_2, d_1 \leftarrow D_1, d_2 \leftarrow D_2]$

Generate candidate policy from sample via value iteration

$\pi_c \leftarrow \text{VI}(z_c)$

Add candidate policy to list of candidate policies

$\Pi_C \xleftarrow{+} \pi_c$

End - M2 Policy Update

Figure 3.4: VIBU-M2 Algorithm

If we wanted the overall policy (as done for VIBU-M1) for plotting or research purposes, we

can compute the overall policy from the candidate policies as follows:

$\pi_{\text{overall}} = \pi_{\emptyset}$

for state $s \in S$ **do**

candidate actions $\mathcal{A}_C = \{a_c = \pi_c[s]\}_{\pi_c \in \Pi_C}$

$\hat{a} = \arg\max_a \sum_{a_c \in \mathcal{A}_C} \mathbb{I}[a_c = a]$

$\pi_{\text{overall}}[s] = \hat{a}$

For each state, we can pick the most common action of all the candidate policies hence the overall policy is the modal action of all candidate policies. Note that for VIBU-M2, we do not need the overall policy as it is computationally cheaper to calculate the action for a given state than to calculate it for every possible state and hence it is not included in the full algorithm.

3.2.2.3 Maximum Discounted Return $\mathcal{M}3$

This method is equivalent to $\mathcal{M}1$ but instead of using V_{avg} as in $\mathcal{M}1$ we can use the element-wise maximum of the different candidate V matrices which can be used to compute the new policy.

$$V_{max\ ij} = \max_c V_{c\ ij}$$

We find that this does not work better than $\mathcal{M}1$ and hence is not investigated further. Practically, taking the maximum distorts the relative value of the elements of the new value matrix and hence does not perform as well.

3.3 Monte-Carlo Dropout Q-Network (MCDQN)

We develop a second method using a Bayesian version of neural network approach that maintains a probability distribution over the Q-values [3]. Bayesian Q-learning suggested by Dearden et al. directly uses continuous probability distributions for the posterior while Bayesian DQN by Azizzadenesheli and Anandkumar explores a conventional DQN [9] with Thompson sampling as well as a double-DQN with the final layer using Bayesian linear regression instead of linear regression. From the Bayesian analogy of Monte-Carlo dropout [6], we opt to use the conventional DQN [9] with Monte-Carlo dropout to generate the distribution over the Q-values.

The key implementation stages of MCDQN are defined below with the full agent implementation present in Listing A.2.

3.3.1 Experience Replay

We convert experience replay defined in Section 2.4.2 into a useable implementation below.

Given a state s, the DQN returns the approximated Q-values for all possible actions as a single vector $\mathcal{Q}(s) \rightarrow [\hat{Q}(s, a_1), ..., \hat{Q}(s, a_n)]$.

Given an experience sample $e_t = (s_t, a_t, r_t, s_{t+1})$, the following is used to create the training data for the model:

$$\begin{aligned} \text{Get current Q-vector} \quad & \underline{q} = \mathcal{Q}(s_t) \\ \text{Update Q-vector using current experience} \quad & \underline{q}[a_t] = r_t + \gamma \max_{a \in \mathcal{A}} \mathcal{Q}(s_{t+1}) \end{aligned}$$

For a minibatch of B experience samples $\mathcal{E} = \{e_1, ..., e_B\}$, we can use this new dataset of B inputs $X = \{s_1, ..., s_B\}$ and B output vectors $Y = \{\underline{q}_1, ..., \underline{q}_B\}$ to fit the model (train the network model.fit(X, Y)).

```
1 agent.experience_replay(batch_size)
```

Listing 3.1: Experience Replay

3.3.2 Candidate Policy Generation

Neural Q-learning has been shown to provide significant advantages over using Q-tables hence it is important to evaluate this method and so we will develop a Q-network for the JCR problem. To use an uncertainty-based approach, we will use Monte-Carlo dropout [6] instead of using an explicit Bayesian method [3].

The input of the neural network is a state vector for which the network returns a Q-vector (a Q-value for each possible action). More specifically, the output of the neural network given a state is $\mathcal{Q}(s) \Rightarrow [\hat{q}(s, a_1), ..., \hat{q}(s, a_n)]$. Therefore we find the corresponding optimal action $\hat{a} = \mathcal{A}[\arg\max_i \mathcal{Q}(s)_i]$ where $\mathcal{A}$ is the set of possible actions.

Therefore for JCR, we use 2 input units for the state vector (consisting of the number of cars in each location) and 11 output units (for each of the possible actions that can be taken) with

the intermediate layer sizes being determined by experimentation such that the optimal policy can be theoretically achieved.

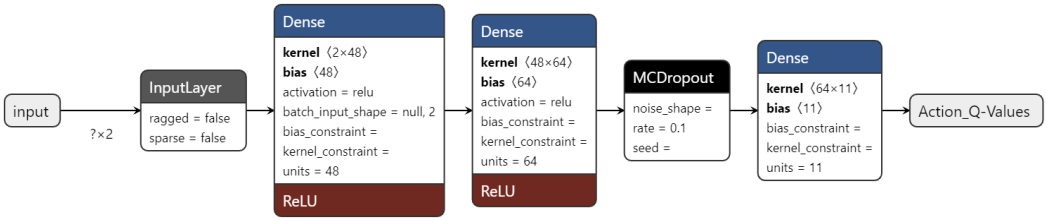


Figure 3.5: MCDQN Architecture [2-48-64-11]

The output layer of the network has linear activation since this is a regression problem for determining the Q-values hence the network is trained using the mean-squared error function. The full network architecture is shown in Figure 3.5.

```
from keras.layers import Dropout
class MCDropout(Dropout):
    def call(self, inputs):
        return super().call(inputs, training=True)
```

Listing 3.2: Monte-Carlo Dropout Class

We implement the MCDropout layer as shown in Listing 3.2 where setting the training parameter to true activates the dropout layer during prediction as well as training. The Monte-Carlo dropout layer will be used to provide us with multiple Q-value estimates. The DQN essentially acts as the posterior distribution of the environment parameters which generates Q-values without explicitly being given the observations of the random variables using only $\underline{s}, a, \underline{s}'$. The observations d_1, r_1, d_2, r_2 which are the demands and returns at each location are implicitly encoded in $\underline{s}, a, \underline{s}'$ where $s_1' = s_1 - a - d_1 + r_1$ and $s_2' = s_2 + a - d_2 + r_2$.

3.3.3 Bayesian Policy Selection

Similarly to VIBU, we need to unify the generated candidate Q-vectors into a single policy. Below we consider the same two methods as outlined before but this time using the neural network outputs which are the Q-value vectors.

3.3.3.1 Mean Q-Value $\mathcal{M}1$

In a similar way to VIBU $\mathcal{M}1$, we can now compute the mean of the N candidate Q-vectors across the actions and then pick the best action which is the action with the largest Q-value in the mean Q-vector.

$$\hat{a} = \arg\max_{a \in \mathcal{A}} \frac{1}{N} \sum_{m=1}^{N} \mathcal{Q}_m(s)$$

3.3.3.2 Thompson Sampling $\mathcal{M}2$

Alternatively in a similar way to VIBU $\mathcal{M}2$, we can calculate an action from each of the Q-value estimates generate by the neural network and pick the most common action from all the candidate

actions $\mathcal{A}_C$ which is equivalent to Thompson sampling.

$$\mathcal{A}_C = \{\arg\max_{a\in\mathcal{A}} \mathcal{Q}_m(s)\}_{m=1}^{N}$$
$$\hat{a} = \arg\max_{a\in\mathcal{A}} \sum_{a_c\in\mathcal{A}_C} \mathbb{I}[a_c = a]$$

3.3.4 MCDQN Algorithm

We expand on the existing Q-network algorithm (Figure 2.4) by simply generating multiple outputs from the Q-network from applying Monte-Carlo dropout. Epsilon-based exploration is neglected and instead $\mathcal{M}1$ or $\mathcal{M}2$ is used instead for action selection using the generated candidate Q-vectors.

assign $\mathcal{E} = \{\}$ empty experience buffer
assign $\mathcal{Q} \leftarrow \theta_{n=0}$ random network weights
$N = 10, L = 20, T = 200, B = 32$
for episode $e \leftarrow 1$ to L **do**
  $s \leftarrow$ reset env
  for step $t \leftarrow 1$ to T **do**
    $\hat{a} =$ [Action Selection Strategy (M1 or M2)]
    $s',\ r,\ \text{done},\ [r_1, r_2, d_1, d_2] \leftarrow$ simulate env $\hat{a}$
    $\mathcal{E} \xleftarrow{+} (s, a, r, s', \text{done})$
    $s = s'$
    if done **do**
      break
    if $|\mathcal{E}| \geq B$ **do**
      $\{\ \underline{e}_j = (s_j, a_j, r_j, s'_j, \text{done}_j)\ \}_{j=1}^{B} \sim \mathcal{E}$ Sample B experiences from $\mathcal{E}$
      $$\{Q_j\}_{j=1}^{B} = \begin{cases} r_j & \text{done}_j = \text{True} \\ r_j + \gamma \max_{a'} \mathcal{Q}(s'_j, a'; \theta_n) & \text{done}_j = \text{False} \end{cases}$$
      $$L(\theta) = \frac{1}{B}\sum_{i=1}^{B}(Q_i - \mathcal{Q}(s_i, a_i; \theta_n))^2$$
      $$\theta_{n+1} = \theta_n - \alpha\nabla L(\theta_n)$$

Figure 3.6: MCDQN Algorithm

The MCDQN policy is updated on every step due to experience replay as once the memory

buffer exceeds the batch size, experience replay occurs on every step. This is in stark contrast to VIBU where the high expense computation means that the policy must be updated every few episodes. The full algorithm is shown in Figure 3.6 and the corresponding simulation code files for MCDQN are shown in Listing A.1 and A.2.

3.4 Comparisons

3.4.1 Estimation

Here we outline how the two estimation techniques (M1 and M2) are different and how we might expect them to compare. The two methods are very similar in that both are computed using some form of the value matrix. For M1, we compute the mean of the Q-value matrix to find the optimal policy while in M2, we use the candidate actions and pick the most common action (Thompson sampling). M1 can be expected to be successful if the Q-values are correctly estimated while M2 will be successful if the selected action is according to the optimal policy. If Q-values are generally correct, averaging them should be remove any outliers however if they are generally incorrect, the uncertainty due to averaging will increase inaccuracy. For a given state, if the policies are generally correct, taking the most common action will be an exploitative approach and will maximise reward but if the policies are generally inaccurate, the process will be more exploratory and may be less effective at exploitation [15]. It is unclear without investigation what estimation method of M1 or M2 will perform better for the JCR as success is also dependent on the characteristics of the MDP as well as the approach chosen (VIBU or MCDQN).

3.4.2 Approach

Here we discuss how the two approaches namely VIBU and MCDQN are different but provide a similar backbone for solving the JCR MDP. While VIBU attempts to learn the model parameters and calculate the policy from the model, MCDQN attempts to learn the policy directly. These approaches are very different and VIBU has an added advantage that if the model parameters are estimate correctly then as long as the model itself is correct, the policy will be optimal. The key aim is to evaluate the two methods for POMDPs and not just the JCR and hence both a model-based and a model-free approach are being investigated. Therefore a key difference to note is that rewards are simulated for MCDQN but calculated for VIBU. While VIBU generates beliefs about the model parameters from experience, MCDQN generates beliefs about the Q-values from experience. Generally non-stationary Q-values pose a challenge in Q-learning convergence and so we may expect VIBU to outperform MCDQN. However we may be able to reach a sufficient policy using MCDQNs with the added benefit of being model-free and hence we perform the investigations below.

Chapter 4

Results & Discussion

We investigate the implemented strategies VIBU and MCDQN for JCR to compare them and evaluate their performance. We compare M1 against M2 to determine which method best solves the exploration-exploitation dilemma while evaluating whether a value-based approach might work better than a model-based approach for POMDPs.

4.1 Definitions

For our investigations, we define the following terms:

Step - A single "day" in the JCR problem during which an action is taken after which a stochastic transition occurs.

Episode - An iteration of the JCR problem until the agent fails or termination occurs for which the inference model is kept the same. This can last an arbitrary amount of steps.

Simulation - A predefined number of episodes that are run for which the inference model is refreshed primarily to test performance.

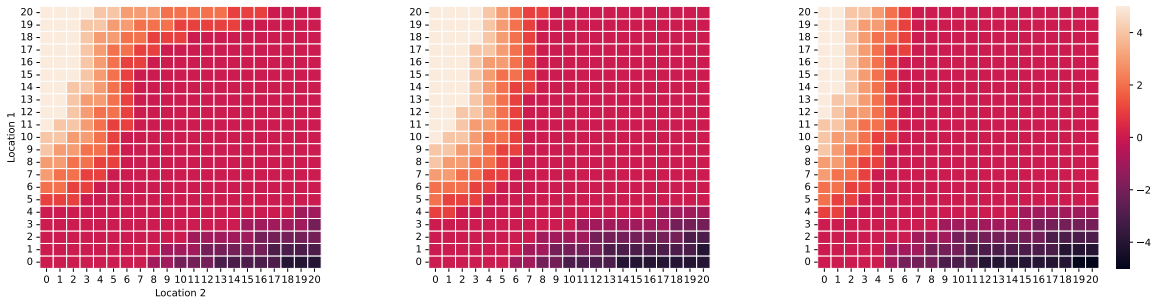
Performance is measured in terms of the total reward per episode over several episodes where each episode is an iteration of JCR until failure or termination. Hence we use the mean and the inter-quartile range (IQR) of the total reward for each episode to measure performance after averaging across episodes over multiple simulations. IQR has been used instead of standard deviation to show the asymmetricity of upside and downside deviation. We should expect performance of the inference model to increase over multiple episodes for a single simulation but more visibly when averaging over multiple simulations.

4.2 Estimation Evaluation

We now compare M1 and M2 for both VIBU and MCDQN to evaluate which method might be preferred while considering the advantages and disadvantages of each estimation technique.

4.2.1 VIBU-M1 vs VIBU-M2

Figure 4.1 shows example policy matrices for JCR with colour-coded actions indicated for each state. Both methods $\mathcal{M}1$ and $\mathcal{M}2$ converge to a similar policy in the same number of episodes as expected (due to the use of value iteration) but $\mathcal{M}1$ converges to a marginally better policy than $\mathcal{M}2$ as shown in Figure 4.1. Both VIBU policies (Figure 4.1b and c) are reflective of the true optimal policy (Figure 4.1a) although not exactly equivalent.



(a) Optimal Policy (b) $\mathcal{M}1$ Policy after 20 episodes (c) $\mathcal{M}2$ Policy after 20 episodes

Figure 4.1: Policy Comparison

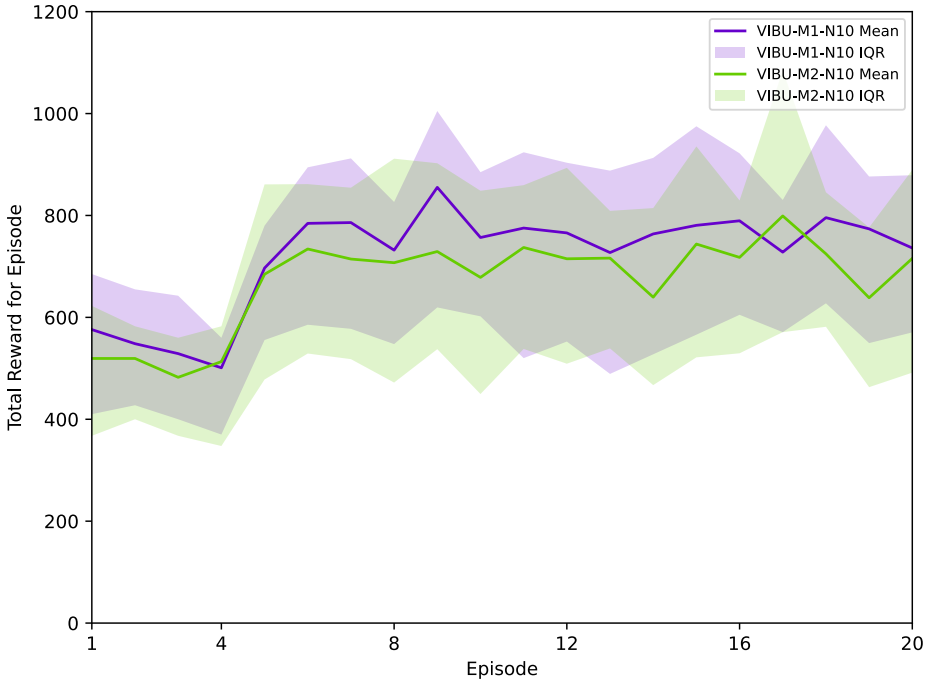


Figure 4.2: Performance of VIBU-M1 vs VIBU-M2 averaged over 60 simulations (N=10 estimates)

We ran the simulation for 20 episodes and generated new policies every 4 episodes for each of the $N = 10$ model parameter observations. Figure 4.2 shows the total reward for each episode averaged over 60 simulations. VIBU-M1 almost has consistently higher mean and quartiles to VIBU-M2 which reflects the generated policy matrices (Fig 4.1).

Convergence of the model posterior often occurs immediately after the first generation process (after the first 4 episodes). However, by sampling the posterior a mixture of candidate policies can be created and effectively selected by generating uncertainty within the environment parameters. In the case of POMDPs, this would be extremely useful since model parameters of non-stationary processes can be effectively estimated and an appropriate policy can be generated.

4.2.2 MCDQN-M1 vs MCDQN-M2

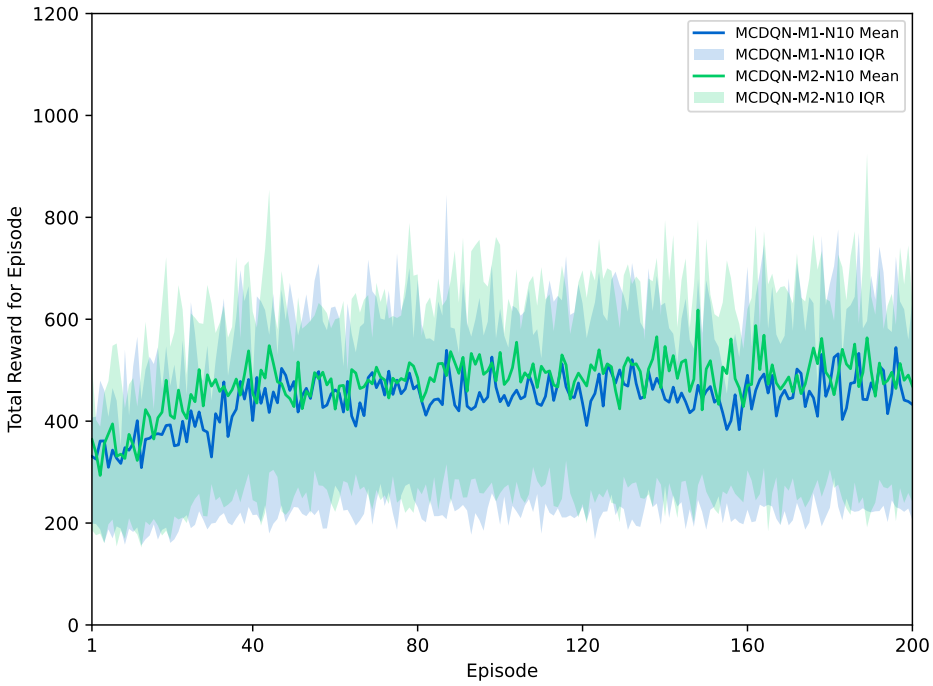


Figure 4.3: Episodic reward performance of MCDQN-M1 vs MCDQN-M2 averaged over 60 simulations (N=10 estimates)

Figure 4.3 shows that MCDQN-M2 consistently outperforms with a higher mean total reward per episode as well as higher quartiles indicating a clear improvement over MCDQN-M1. The averaging of Q-values in MCDQN-M1 leads to a more uniform distribution across the Q-vector hence being more exploratory and attempting inaccurate policies. MCDQN-M2 selects the most common action and hence only reinforces the belief if the consequential reward is high otherwise the network will modify the policy. Therefore the Thompson sampling approach is more exploitative which is a major reason why MCDQN-M2 performs well for selecting model parameters (analogous to the multi-armed bandit problem [13]).

The loss graph in Figure 4.4 shows that the neural network is learning and as the policy changes by exploration-exploitation during experience replay, the loss begins to plateau. Each experience replay corresponds to a batch of $B = 32$ experience samples which are the days in JCR. To notice the huge reduction in loss within the first 10 batches for MCDQN-M2 is extremely encouraging as this shows applicability for POMDPs even with Q-learning. Smaller batch sizes do still converge albeit more slowly and with higher variance. The vast improvement of MCDQN-M2 over MCDQN-M1 is again clearly reflected by the convergence of the training supporting the reward performance plot in Figure 4.3.

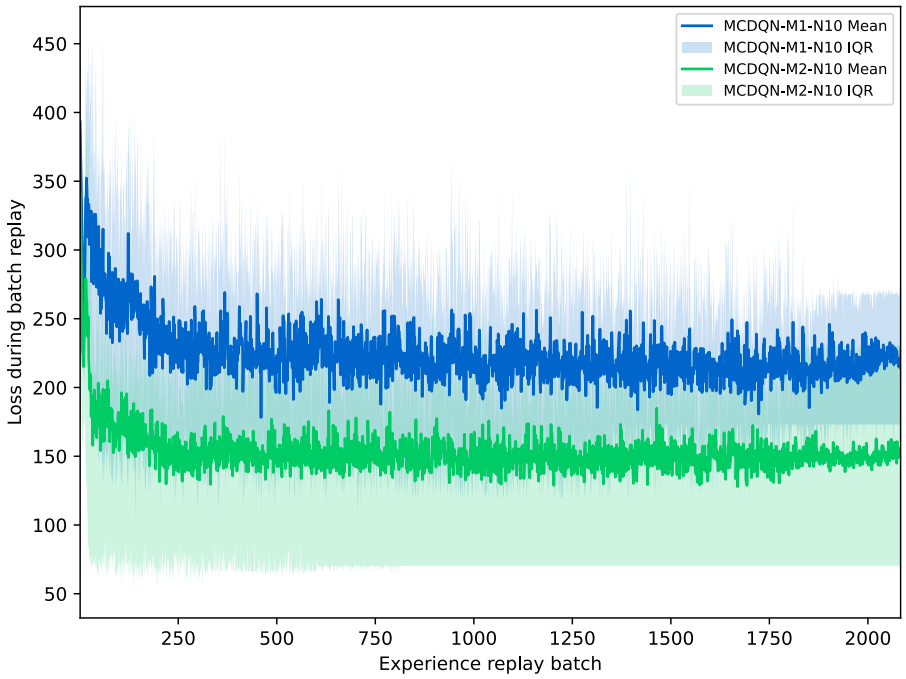


Figure 4.4: MCDQN MSE loss for 200 episodes averaged over 60 simulations (N=10 estimates)

4.2.3 Overview

From our investigation, we find that M2 performs significantly better than M1 for MCDQN but M1 performs marginally better than M2 for VIBU for the JCR problem. These results indicate that the MCDQN is not very accurate at learning the Q-values themselves but is able to determine action selection hence the poor performance for M1 and better performance for M2. The opposite is true for VIBU; since VIBU learns the true Q-function for a given set of model parameters by value iteration, picking the most common action (M2) out of a set of candidate policies only increases uncertainty since not all policies are likely to be correct hence action selection accuracy is lower.

VIBU-M1 performs marginally better than VIBU-M2 since all the candidate policies are relatively similar as the model parameters do not have high ranges. If the model parameters were to have much more variation by being scaled up (along with the maximum number of cars per location) where the Poisson means of the demands and returns were $\{Z+3, Z+4, Z+3, Z+2\}$ where Z was a large number such as 50 (instead of $Z = 0$ as in the default parameters), VIBU-M1 would not perform as well as VIBU-M2. VIBU-M2 would be able to generate a variety of candidate policies and be able to exploit and explore effectively at the same time. This is clearly shown by the performance improvement of MCDQN-M2 over MCDQN-M1. This shows that Thompson sampling is a really effective method of exploration and exploitation for large state and action spaces.

4.3 Approach Evaluation

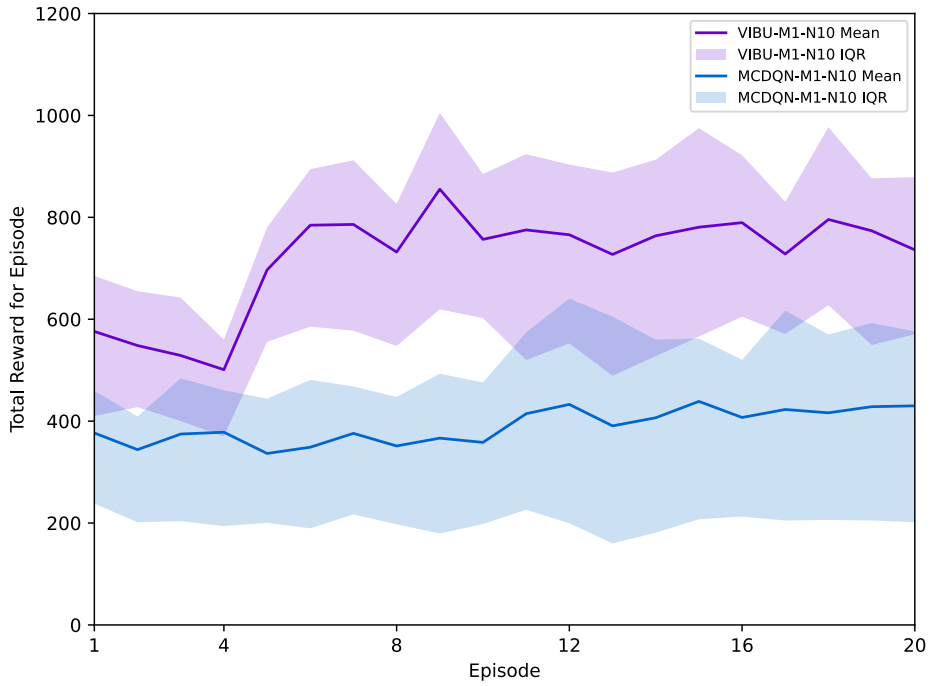
We now compare the results of the two different methods to each other specifically showing the difference between the performance of VIBU and MCDQN. It is very important to note that both these methods investigate 2 different approaches namely VIBU attempting to determine the policy by finding the model and MCDQN attempting to determine the policy by learning the value function. However the objective is to attain sufficient applicability to POMDPs and hence it is important to consider which method may prove more fruitful for generating good policies for POMDPs in real-world scenarios.

4.3.1 VIBU vs MCDQN

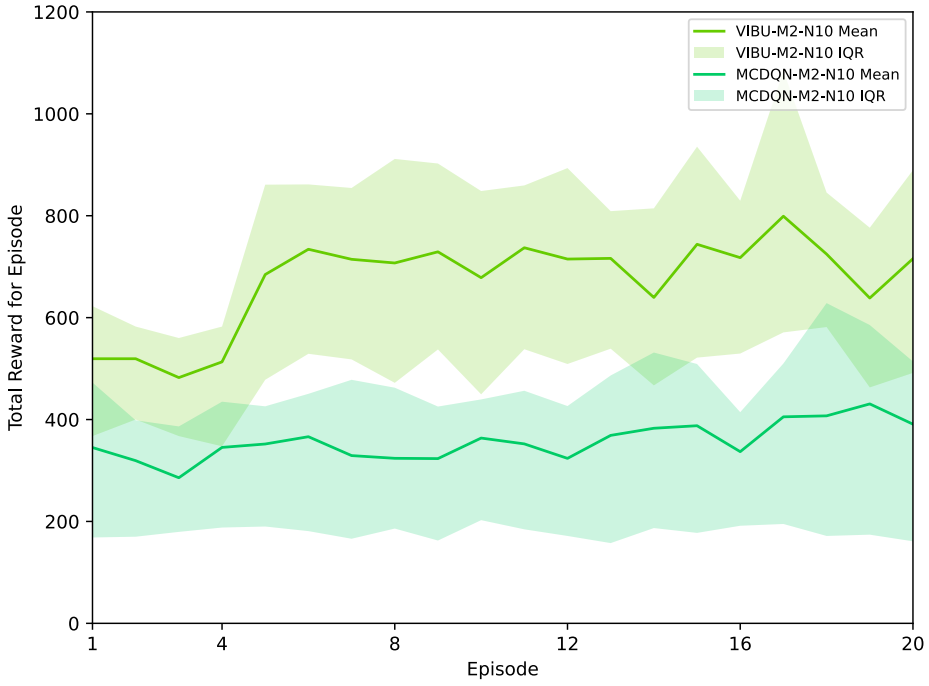
For both methods M1 and M2, VIBU provides a significant advantage in terms of performance over MCDQN especially with applicability to POMDPs. Figure 4.5 shows that VIBU outperforms MCDQN by a significant margin where the lower quartile of VIBU is almost consistently higher than the upper quartile of MCDQN. As from Figure 4.1, the policies are generally near optimal and hence this is as expected. However the significance of VIBU is shown by the fast rate of convergence and consistency in total reward even when the policies are changing regularly.

4.3.2 Overview

The fast convergence rate of VIBU with very few experience samples offers a significant advantage over MCDQN. This is specifically because the iterative process of model generation-evaluation-improvement is directly applicable to many real-world scenarios and value iteration provides the optimal policy for a given model. Nonetheless, MCDQN also offers a few important advantages over VIBU such as being scalable to continuous state-space problems and learning the policy directly without constructing a model. Model construction is not always feasible especially for non-stationary processes where a heuristic policy could work better. For JCR, VIBU is generally good when variance is high but it assumes the model design is appropriate. Some games/processes may have starkly different policies for sub-optimal estimates.



(a) VIBU-M1 vs MCDQN-M1



(b) VIBU-M2 vs MCDQN-M2

Figure 4.5: Performance of VIBU vs MCDQN averaged over 60 simulations (N=10 estimates)

4.4 Estimates Exploration

To analyse performance and check consistency with a larger number of estimates, we investigate the previous strategies with more samples.

4.4.1 VIBU Parameter Estimates

During our experimentation with VIBU for JCR, we find that more estimates do not provide additional benefit for determining the model provided that a sufficient number of estimates are used. Using hypothesis testing, we can determine that the 95% confidence interval for the mean of a Poisson random variable using N samples is:

$$\hat{\lambda} \pm \phi^{-1}(0.975)\sqrt{\frac{\hat{\lambda}}{N}}$$

$\hat{\lambda}$ is the mean of the samples. $\phi^{-1}(p)$ is the inverse CDF of a standard normal distribution. For $N = 10$ samples, the confidence interval is $\hat{\lambda} \pm \underbrace{\frac{\phi^{-1}(0.975)}{\sqrt{10}}}_{\text{0.6198 to 4dp}} \sqrt{\hat{\lambda}}$.

Num Estimates (N)	10	20	30	40
Confidence Scale	0.6198	0.4383	0.3578	0.3099

Table 4.1: Confidence scaling values $\frac{1.96}{\sqrt{N}}$ with N estimates

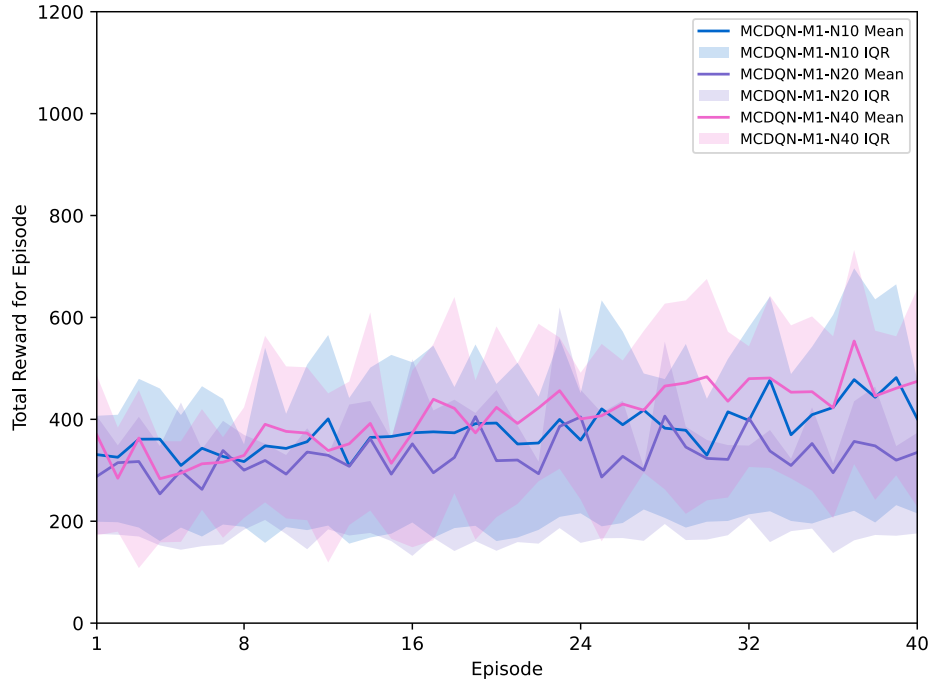
From Table 4.1, we see that more samples could be marginally beneficial up to a maximum of 30 estimates after which there is very little benefit in using more estimates as the confidence value reduces with $\frac{1}{\sqrt{N}}$. Therefore computing more estimates is prohibitively more expensive when computing candidate policies. In the case of POMDPs, if the assumed model is incorrect increasing the number of estimates provides no added advantage as the value-iterated policy would be sub-optimal. However, if we assume that the POMDP model is correct, increasing the estimates will provide an improved strategy as outlier samples will have reduced impact on the model posterior distribution.

4.4.2 MCDQN Q-Vectors

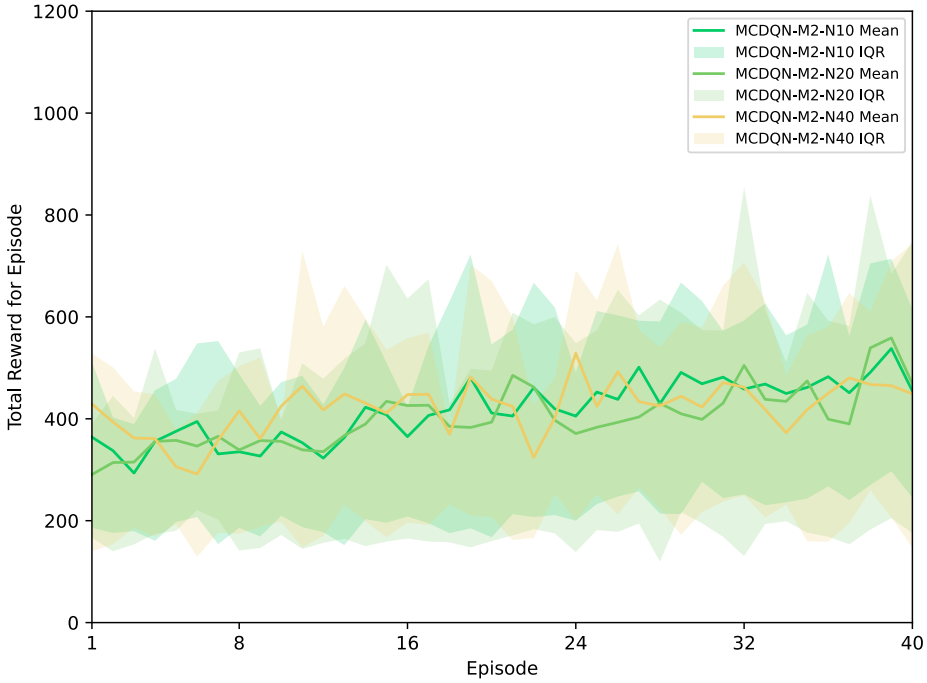
We further explore the number of estimates for MCDQN to determine whether more estimates helps improve policy convergence.

Figure 4.6 shows that the number of estimates help to speed up initial convergence to a temporally-local policy which is actually very beneficial for POMDPs but we must note that in the long term, more estimates provide very little benefit in finding an improved policy which is the same verdict as for VIBU.

By comparing across MCDQN-M1 and MCDQN-M2 (Figure 4.6), we see that M2 also always outperforms M1 with M2 having a consistently higher mean and upper quartile for all number of



(a) MCDQN Method 1



(b) MCDQN Method 2

Figure 4.6: Performance of MCDQN-M1 and MCDQN-M2 averaged over 60 simulations for N=10, 20, 40 estimates

estimates. This confirms our previous analysis (Section 4.2.2) which shows that MCDQN-M2 outperforms MCDQN-M1.

4.4.3 Overview

In both cases for VIBU and MCDQN, the JCR MDP does not require a significant number of parameter estimates. However it must be noted that the JCR requires very few estimates in the first place to converge to the true policy. In the case of a non-stationary process such as POMDPs, more samples would definitely provided added benefit. For POMDPs, the greatest drawback would be the computational cost of generating more candidate policies from estimates.

In terms of computational cost, MCDQN outperforms VIBU significantly with MCDQN-M1 and MCDQN-M2 both having similar performance costs while VIBU-M1 having a lower cost than VIBU-M2. VIBU-M1 only needs to iterate through all the states once to compute the overall policy unlike VIBU-M2 which has to compute a policy for every estimate. For a large state-space, a larger number of estimates would be required and hence VIBU-M2 would become increasingly expensive.

The time complexity of the value iteration function (Figure 2.3) is $\mathcal{O}(|S|^2|A|K + |S|^2|A|)$ (excluding the effect of the discount factor) [14]. Therefore for VIBU-M1, only the overall policy is computed therefore having a time complexity of $\mathcal{O}(|S|^2|A|KN + |S|^2|A|)$ (value iteration is performed N times but policy is computed only once) while VIBU-M2 would have a time complexity of $\mathcal{O}(|S|^2|A|KN + |S|^2|A|N)$ (both value iteration and policy are computed N times) where $|S|$ is the number of states, $|A|$ is the number of actions, K is the number of value iterations and N is the number of estimates.

Chapter 5

Conclusion

Computational complexity poses a difficult challenge in experimentation and development with major drawbacks to using more intensive algorithms. Value-based methods can be applied to simple problems to achieve great performance. Hence value iteration for discrete state spaces is optimal and works well when dimensionality is low but the same does not necessarily apply with Q-learning. Model-based approaches work indisputably well where the main challenge is model estimation and while model-free approaches are good at estimation, convergence is difficult and expensive.

5.1 Findings

From our investigations involving the 4 strategies MCDQN-M1, MCDQN-M2, VIBU-M1 and VIBU-M2, we find that MCDQN is generally good at finding an appropriate policy however these are not optimal and would require a significantly larger network as well as more experience for a harder problem. It must be noted that the value-based approach for finding the policy using MCDQN is computationally much cheaper than VIBU where value iteration is directly performed. If the policies generated by MCDQN are sufficiently good, it may be more effective to create an adaptive value-based approach rather than a model-based approach. Nonetheless for the JCR problem, VIBU generates significantly better policies with high confidence. Now considering the method of policy recombination, M2 using Thompson sampling performs generally better than M1. Comparing VIBU-M1 and VIBU-M2, the preferred approach is VIBU-M2 even though VIBU-M1 converges to a marginally better policy. When the candidate policies are localised in the policy space meaning that all the candidate policies are very similar, VIBU-M1 effectively generates the averaged policy which is better than VIBU-M2 only if the candidate policies are close to optimal. This may not necessarily be the case for POMDPs as belief-based policies can have high variance. Thompson sampling (M2) is extremely effective at exploiting assumed beliefs and exploring uncertain state-action trajectories as shown by the performance improvement of MCDQN-M2 over MCDQN-M1 as well as VIBU-M2 being comparably as good as VIBU-M1.

We consider a relatively small number of estimates for generating the policies due to the high computational cost. However we also find that the impediment to producing good policies is the method used to generate the policies rather than the number of samples or even the effectiveness of the sampling method. This is more apparent with VIBU where both methods M1 and M2 perform similarly well due to value iteration being used for both which finds the optimal policy for a given model. Whereas with MCDQN, there is a much clearer distinction as to which policy

recombination method performs better since Q-learning is ineffective with few iterations and it is clear to see that M2 with Thompson sampling has a much better performance than M1. Note that this only applies in the case of MDPs where value iteration can feasibly be used but does not apply to POMDPs. For POMDPs, the policy estimation as well as the sampling technique will have equally significant impacts on performance since the optimal policy is not tractably computable.

5.2 Future Work

From our study, we find that MCDQN is a good contender for further development. However generally the DQN-based approach does not work well when real-experience is limited. We could assume a model from static experience and then develop a DQN for the simulated version which could be applied sub-optimally to the real-scenario however this approach of model iteration is expensive and high variance models would require frequent updating which is not always feasible. Monte-Carlo dropout on the other hand is a useful Bayesian tool for neural networks. It may be possible to use a model-based approach using a neural network method for approximating the posterior where the Bayesian conjugate for the model parameters are unknown. Since our project mainly considers MDPs where optimal policies are directly computable, using value iteration was a medium for evaluating the methods of combining policies. However VIBU can also be generally applied to POMDPs by replacing value iteration with some approximator for the Q-value distribution. Regarding compatibility with continuous state and action spaces, value iteration in VIBU would be ineffective while MCDQN could be theoretically extended. Hence the use of mixture models or Gaussian processes may be effective for certain problems. Overall, the choice of algorithm for translating a POMDP model to a policy is a complex problem that does not have a general solution. Hence constructing belief-based Q-value or policy approximators directly might be more effective than constructing models and then generating policies. This advocates for policy-based approaches over model-based approaches especially when mediocre policies are acceptable over mediocre models in the real-world. However for problems where the model is well-defined but uncertain and the optimal policy is computable, VIBU is extremely effective and performs well when used with Thompson sampling with evidence to suggest good performance for complex models too.

Bibliography

[1] John Asmuth, Lihong Li, Michael L. Littman, Ali Nouri, and David Wingate. A bayesian sampling approach to exploration in reinforcement learning, 2012. URL https://arxiv.org/abs/1205.2664.

[2] Kamyar Azizzadenesheli and Animashree Anandkumar. Efficient exploration through bayesian deep q-networks, 2018. URL https://arxiv.org/abs/1802.04412.

[3] Richard Dearden, Nir Friedman, and Stuart J. Russell. Bayesian q-learning. In *AAAI/IAAI*, 1998.

[4] Richard Dearden, Nir Friedman, and David Andre. Model-based bayesian exploration. In *UAI*, 1999.

[5] Gabriel Dulac-Arnold, Daniel J. Mankowitz, and Todd Hester. Challenges of real-world reinforcement learning. *CoRR*, abs/1904.12901, 2019. URL http://arxiv.org/abs/1904.12901.

[6] Yarin Gal and Zoubin Ghahramani. Dropout as a bayesian approximation: Representing model uncertainty in deep learning, 2016.

[7] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[8] Lingheng Meng, Rob Gorbet, and Dana Kulic. Memory-based deep reinforcement learning for POMDP. *CoRR*, abs/2102.12344, 2021. URL https://arxiv.org/abs/2102.12344.

[9] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013. URL http://arxiv.org/abs/1312.5602.

[10] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015. doi: 10.1038/nature14236. URL https://doi.org/10.1038/nature14236.

[11] Andrew W. Moore and Christopher G. Atkeson. Prioritized sweeping: Reinforcement learning with less data and less time. *Machine Learning*, 13(1):103–130, October 1993. doi: 10.1007/bf00993104. URL https://doi.org/10.1007/bf00993104.

[12] Ian Osband and Benjamin Van Roy. Why is posterior sampling better than optimism for reinforcement learning?, 2016. URL https://arxiv.org/abs/1607.00215.

[13] Daniel J. Russo, Benjamin Van Roy, Abbas Kazerouni, Ian Osband, and Zheng Wen. A tutorial on thompson sampling. *Foundations and Trends® in Machine Learning*, 11(1): 1–96, 2018. doi: 10.1561/2200000070. URL https://doi.org/10.1561/2200000070.

[14] Aaron Sidford, Mengdi Wang, Xian Wu, Lin F. Yang, and Yinyu Ye. Near-optimal time and sample complexities for solving discounted markov decision process with a generative model. *arXiv: Optimization and Control*, 2019.

[15] Malcolm Strens. A bayesian framework for reinforcement learning. In *In Proceedings of the Seventeenth International Conference on Machine Learning*, pages 943–950. ICML, 2000.

[16] Richard S. Sutton. Integrated architectures for learning, planning, and reacting based on approximating dynamic programming. In *Machine Learning Proceedings 1990*, pages 216–224. Elsevier, 1990. doi: 10.1016/b978-1-55860-141-3.50030-4. URL https://doi.org/10.1016/b978-1-55860-141-3.50030-4.

[17] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA, 2018. ISBN 0262039249.

[18] Tao Wang, Daniel Lizotte, Michael Bowling, and Dale Schuurmans. Bayesian sparse sampling for on-line reward optimization. In *Proceedings of the 22nd international conference on Machine learning - ICML '05*. ACM Press, 2005. doi: 10.1145/1102351.1102472. URL https://doi.org/10.1145/1102351.1102472.

[19] Aaron Wilson, Alan Fern, Soumya Ray, and Prasad Tadepalli. Multi-task reinforcement learning. In *Proceedings of the 24th international conference on Machine learning - ICML '07*. ACM Press, 2007. doi: 10.1145/1273496.1273624. URL https://doi.org/10.1145/1273496.1273624.

[20] Pengfei Zhu, Xin Li, and Pascal Poupart. On improving deep reinforcement learning for pomdps. *CoRR*, abs/1704.07978, 2017. URL http://arxiv.org/abs/1704.07978.

Appendix A

MCDQN

```
for ep in range(1, episodes+1):
    done = False
    step = 0
    state = env.reset()
    rewards = []

    while True:
        step += 1

        action = agent.action(state)
        new_state, reward, done, info = env.step(action)
        agent.store(state, action, reward, next_state, done)

        state = new_state
        rewards.append(reward)

        if done or step > 200:
            total_rewards = sum(rewards)
            total_steps = step
            if ep % 1 == 0:
                print(f'Episode: {ep} Reward: {total_rewards} Steps: {total_steps}')
                all_rewards.append(total_rewards)
                all_steps.append(total_steps)
            break

        # Experience Replay
        if len(agent.memory) > batch_size:
            agent.experience_replay(batch_size)
```

Listing A.1: MCDQN Simulation Loop

```
class MCDropout(Dropout):
    def call(self, inputs):
        return super().call(inputs, training=True)

class MCDQNAgent():
    def __init__(self, states, actions, alpha, gamma, method_type=2, num_estimates=10,
     train_epochs=20):
        self.nS = states
        self.actions = actions
        self.nA = len(actions)
        self.memory = deque([], maxlen=2500)
        self.alpha = alpha
        self.gamma = gamma

        self.model = self.build_model()
        self.loss = []
        self.method_type = method_type
        self.num_estimates = num_estimates
        self.train_epochs = train_epochs

    def build_model(self):
        model = keras.Sequential(name="MCDQN")
        model.add(Dense(48, input_dim=self.nS, activation='relu'))
        model.add(Dense(64, activation='relu'))
        model.add(MCDropout(0.1))
        model.add(Dense(self.nA, activation='linear', name="Action_Q-Values"))
        model.compile(loss='mean_squared_error', optimizer=keras.optimizers.Adam(learning_rate=
     self.alpha))
        # model.save('2-48-64-11.h5')
        return model

    def action(self, state):
        states = np.tile(np.array(state), (self.num_estimates, 1))
```

```
32          if self.method_type == 1:
33              # Method 1
34              avg_action_vals = self.model.predict(states).mean(axis=0)
35              best_action = self.actions[np.argmax(avg_action_vals)]
36          elif self.method_type == 2:
37              # Method 2
38              act_indexes = np.argmax(self.model.predict(states), axis=1)
39              best_action = self.actions[Counter(act_indexes).most_common(1)[0][0]]
40          else:
41              # Default single estimate
42              best_action = self.actions[np.argmax(self.model.predict(np.reshape(state, [1, self.nS
        ]))))]
43          return best_action
44
45      def store(self, state, action, reward, nstate, done):
46          # Get index of action
47          action_index = self.actions.index(action)
48          # Store the experience in memory
49          self.memory.append( (state, action_index, reward, nstate, done) )
50
51      def experience_replay(self, batch_size):
52          # Randomly sample from memory
53          minibatch = np.array(random.sample( self.memory, batch_size ))
54          # Convert to numpy for speed by vectorization
55          x, y = [], []
56          st = np.zeros( (0,self.nS) ) # States
57          nst = np.zeros( (0,self.nS) ) # Next States
58          for i in range(len(minibatch)): # Creating the state and next state np arrays
59              st = np.append( st, minibatch[i,0], axis=0)
60              nst = np.append( nst, minibatch[i,3], axis=0)
61          # Speedup by using predict on the entire batch
62          st_predict = self.model.predict(st)
63          nst_predict = self.model.predict(nst)
64
65          for index, (state, action_index, reward, nstate, done) in enumerate(minibatch):
66              x.append(state)
67              # Predict from state
68              nst_action_predict_model = nst_predict[index]
69              if done == True: # Terminal
70                  target = reward
71              else: # Non terminal
72                  target = reward + self.gamma * np.max(nst_action_predict_model)
73              target_f = st_predict[index]
74              target_f[action_index] = target
75              y.append(target_f)
76
77          # Reshape for Keras Fit
78          x_reshape = np.array(x).reshape(batch_size,self.nS)
79          y_reshape = np.array(y)
80
81          # Epochs is the number of training iterations
82          epoch_count = self.train_epochs
83          hist = self.model.fit(x_reshape, y_reshape, epochs=epoch_count, verbose=0)
84          # Graph Losses
85          for i in range(epoch_count):
86              self.loss.append( hist.history['loss'][i] )
```

Listing A.2: MCDQN Agent Python

Appendix B

VIBU

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from env import JCREnv
from collections import Counter
from scipy.stats import poisson, gamma
import multiprocessing as mp
#import torch.multiprocessing as mp
import csv

def value_iteration(env, max_iterations=100, gam=0.9):
    V = {s: 0 for s in env.states}
    theta = 0.01  # V(s) delta stopping threshold
    max_cars = env.max_cars

    for k in range(max_iterations):
        delta = 0
        V_old = V.copy()
        V = {s: 0 for s in env.states}

        for state in env.states:
            v_best = -1000  # used to find best value
            # constrain actions to possible actions
            max_a = min(5, state[0], max_cars-state[1])
            min_a = max(-5, -state[1], -(max_cars-state[0]))
            # V[s] = max_a (    _s   ' P(s'|s,a) (R(s,a,s')+     V_{k-1}[s']) )
            for a in range(min_a, max_a+1):
                t_model, r_model = env.get_transition_model(state, a)
                v_new = 0
                for s_prime in t_model:
                    p = t_model[s_prime]
                    r = r_model[s_prime]
                    # must use previous iteration's V(s): V_old(s)
                    v_new += p * (gam * V_old[s_prime] + r)
                v_best = max(v_best, v_new)
            V[state] = v_best
            delta = max(delta, abs(V[state] - V_old[state]))
        # until   s       = max_s |V_k[s]-V_{k-1}[s]| <
        print('Iteration {}: max delta = {:.2f}'.format(k, delta))
        if delta < theta:
            break
    return V

def get_policy(env, V, gam=0.9):
    policy = {s: 0 for s in env.states}
    max_cars = env.max_cars
    for state in env.states:
        best_v = -1000
        max_a = min(5, state[0], max_cars-state[1])
        min_a = max(-5, -state[1], -(max_cars-state[0]))
        for a in range(min_a, max_a+1):
            t_model, r_model = env.get_transition_model(state, a)
            v = 0
            for s_prime in t_model:
                p = t_model[s_prime]
                r = r_model[s_prime]
                v += p * (gam * V[s_prime] + r)
            if v > best_v:
                policy[state] = a
                best_v = v
    return policy
```

Listing B.1: VIBU Common Functions Python

```
if __name__=="__main__":
```

```
2
3       rent_lambda_1 = 3
4       rent_lambda_2 = 4
5       return_lambda_1 = 3
6       return_lambda_2 = 2
7
8       method_type = 1
9       episodes = 20
10      num_estimates = 10
11
12      sims = 20
13      for sim in range(1, sims+1):
14          print(f"=== SIMULATION {sim} ===")
15          all_steps = []
16          all_rewards = []
17
18          env = JCREnv(rent_lambda_1, rent_lambda_2, return_lambda_1, return_lambda_2)
19          policy = {s: 0 for s in env.states}
20          candidate_policies = [policy]
21
22          # These are initialised as the priors (Gamma(1, 1)) # Equivalently as 1 sample with a
        value of 1
23          posteriors = [1, 1, 1, 1] # store alpha / beta to avoid overflow
24          posterior_counts = [1, 1, 1, 1] # beta parameter
25
26          for ep in range(1, episodes+1):
27              done = False
28              step = 0
29              state = env.reset()
30
31              rewards = []
32              earnings = [0]
33              while True:
34                  step += 1
35                  action = policy[state]
36                  new_state, reward, done, info = env.step(action)
37
38                  # Update the posterior
39                  samples = info["transition"]
40                  for pid in range(len(posteriors)):
41                      posteriors[pid] = (posteriors[pid] * posterior_counts[pid] + samples[pid])/(
        posterior_counts[pid]+1)
42                      posterior_counts[pid] += 1
43
44                  state = new_state
45                  rewards.append(reward)
46
47                  if done or step > 200:
48                      total_steps = step
49                      if ep % 1 == 0:
50                          print('Episode: {} Reward: {} Steps Taken: {}'.format(
51                              ep, sum(rewards), total_steps))
52                          all_rewards.append(sum(rewards))
53                      break
54
55              if ep % 4 == 0:
56                  candidate_policies = []
57                  pm_rn1, pm_ret1, pm_rn2, pm_ret2 = posteriors
58
59                  # Multi-processor version - Posterior Samples
60                  N = num_estimates
61                  pool = mp.Pool(mp.cpu_count())
62
63                  sample_estimates = []
64                  for p_est in range(N):
65                      estimates = []
66                      for pid in range(len(posteriors)):
67                          # a is alpha, scale is 1/beta
68                          a, scale = posteriors[pid] * posterior_counts[pid], 1/posterior_counts[pid
        ]
69                          estimate = gamma.rvs(a=a, scale=scale)
70                          estimates.append(estimate)
71                      sample_estimates.append(estimates)
72                  #print(sample_estimates)
73
74                  # Get value matrices
75                  Vcands = pool.map(val_iter_mp_m1, sample_estimates)
76                  # Average value matrices
77                  Vavg = {s: 0 for s in env.states}
78                  for st in env.states:
79                      Vavg[st] = np.mean([Vc[st] for Vc in Vcands])
```

```
                # Get policy by picking action for highest average value over all actions
                policy = get_policy(env, Vavg)

        with open(f'./vibu/m{method_type}/ep{episodes}-N{num_estimates}/full_data.csv','a', newline='') as fd:
            write = csv.writer(fd)
            write.writerow(all_rewards)
```

Listing B.2: VIBU Method-1 Python

```
if __name__=="__main__":

    rent_lambda_1 = 3
    rent_lambda_2 = 4
    return_lambda_1 = 3
    return_lambda_2 = 2

    method_type = 2
    episodes = 20
    num_estimates = 10

    sims = 20
    for sim in range(1, sims+1):
        print(f"=== SIMULATION {sim} ===")
        all_steps = []
        all_rewards = []

        env = JCREnv(rent_lambda_1, rent_lambda_2, return_lambda_1, return_lambda_2)
        policy = {s: 0 for s in env.states}
        candidate_policies = [policy]

        # These are initialised as the priors (Gamma(1, 1)) # Equivalently as 1 sample with a value of 1
        posteriors = [1, 1, 1, 1] # store alpha / beta to avoid overflow # Note that this is also the gamma prior
        posterior_counts = [1, 1, 1, 1] # beta parameter

        for ep in range(1, episodes+1):
            done = False
            step = 0
            state = env.reset()

            rewards = []
            earnings = [0]
            while True:
                step += 1
                candidate_actions = [policy[state] for policy in candidate_policies]
                action = Counter(candidate_actions).most_common(1)[0][0]
                new_state, reward, done, info = env.step(action)

                # Update the posterior
                samples = info["transition"]
                for pid in range(len(posteriors)):
                    posteriors[pid] = (posteriors[pid] * posterior_counts[pid] + samples[pid])/(posterior_counts[pid]+1)
                    posterior_counts[pid] += 1

                state = new_state
                rewards.append(reward)

                if done or step > 200:
                    total_steps = step
                    if ep % 1 == 0:
                        print('Episode: {} Reward: {} Steps Taken: {}'.format(
                            ep, sum(rewards), total_steps))
                        all_rewards.append(sum(rewards))
                    break

            if ep % 4 == 0:
                candidate_policies = []
                pm_rn1, pm_ret1, pm_rn2, pm_ret2 = posteriors

                # Multi-processor version - Posterior Samples
                N = num_estimates
                pool = mp.Pool(mp.cpu_count())

                sample_estimates = []
                for p_est in range(N):
                    estimates = []
                    for pid in range(len(posteriors)):
```

```
68                          a, scale = posteriors[pid] * posterior_counts[pid], 1/posterior_counts[pid
    ]
69                          estimate = gamma.rvs(a=a, scale=scale)
70                          estimates.append(estimate)
71                      sample_estimates.append(estimates)
72                  #print(sample_estimates)
73                  candidate_policies = pool.map(val_iter_mp_m2, sample_estimates)
74
75          with open(f'./vibu/m{method_type}/ep{episodes}-N{num_estimates}/full_data.csv','a',
    newline='') as fd:
76              write = csv.writer(fd)
77              write.writerow(all_rewards)
```

Listing B.3: VIBU Method-2 Python