Statistical Techniques in Robotics (16-831, S22) Lecture #14 (Monday, March 14)

# Sequence Feedback Learning Problems: MDP

*Lecturer: Kris Kitani* *Scribes: Xuanbai Chen, Jia Shi (Group C)*

## 1 Review

In the last lecture, we talked about Thompson Sampling, EXP3 and EXP4 algorithms.

### 1.1 Thompsons Sampling

The step of the Thompsons Sampling is as follows: (1).Maintain a running estimate of the prior distribution hyper-parameter by observing the rewards. (2).Select the best arm by sampling from the estimated posterior distribution. The algorithm is shown below:

**Algorithm 1** Thompsons Sampling

**for** $t = 1, \cdots, T$ **do**
  $\theta_k \sim p(;\alpha_k, \beta_k) \quad \forall k$ sample from posterior
  $a_{\hat{k}}^{(t)} = argmax_k \mathbb{E}_{p(r|a_k,\theta_k)}[r \mid a_k, \theta_k]$
  RECEIVE$(r^{(t)})$ get sampled reward
  $p(\theta_{\hat{k}} \mid h_{\hat{k}}) \propto p(r^{(t)} \mid a_{\hat{k}}^{(t)}, \theta_{\hat{k}}) p(\theta_{\hat{k}} \mid h_{\hat{k}})$ update posterior
**end for**

### 1.2 Thompsons Sampling: Beta-Bernoulli Bandit

If we set that the beta distribution is the conjugate prior of the Bernoulli distribution, we can obtain the Bern-Beta Thompsons Sampling. In this way, the posterior is also beta distributed and it is super easy to compute.

**Algorithm 2** Bern-Beta Thompsons Sampling

**for** $t = 1, \cdots, T$ **do**
  $\theta_k \sim p(;\alpha_k, \beta_k) \quad \forall k$ sample from posterior
  $a_{\hat{k}}^{(t)} = argmax_k \mathbb{E}_{p(r|a_k,\theta_k)}[r \mid a_k, \theta_k]$
  RECEIVE$(r^{(t)})$ get sampled reward
  $\alpha_{\hat{k}} = \alpha_{\hat{k}} + r^{(t)}$ update posterior
  $\beta_{\hat{k}} = \beta_{\hat{k}} + 1 - r^{(t)}$ update posterior
**end for**

It is a no-regret algorithm with the regret bound of:

$$BR \leq O(\sqrt{TK \log T})$$

### 1.3 EXP3 - Exponential-Weight Update algorithm for Exploration and Exploitation

EXP3 is designed for sampling action in context-free adversarial environment. It works by passing in a modified loss vector $\hat{l}_t$ to the experts algorithm EG and by updating the weights of selected action, it manage to obtain the optimal parameter. The algorithm is listed as below.

**Algorithm 3** EXP3($\gamma \in [0, 1]$)

1: $\mathbf{w}^{(1)} \leftarrow \{\mathrm{w}_k^{(1)} = 1\}_{k=1}^K$ weights over actions
2: **for** $t = 1, \cdots, T$ **do**
3: $\mathbf{p}^{(t)} = \frac{\mathbf{w}^{(t)}}{\sum_k w_k^{(t)}}$ probability over actions
4: k $\sim$ MULTINOMIAL($\mathbf{p}^{(t)}$) take and draw action
5: $a^{(t)} = a_k$
6: RECEIVE($r^{(t)} \in [0, 1]$) get reward
7: $w_k^{(t+1)} = w_k^{(t)} e^{\gamma \frac{r^{(t)}}{p_k^{(t)}}}$ update weight for one arm
8: **end for**

The EXP3 is a no regret algorithm by setting

$$\gamma = \sqrt{\frac{\log K}{TK}},$$

the regret bound is

$$R \leq O(\sqrt{TK \log K}).$$

### 1.4 EXP4 - Exponential-Weight Update algorithm for Exploration and Exploitation with Experts

EXP4 algorithm is designed for contextual Multi-Arm Bandit problem. It keep a probability distribution, which we will denote by $Q_t$, over the experts and use this to come up with the next action. Once the action is chosen, we can use our favorite reward estimation procedure to estimate the rewards for all the actions, which can then be used to estimate how much total reward the individual experts would have made so far, which in the end can be used to update $Q_t$. Context means the prior information of the action at the beginning of the round, before it needs to select its action. The algorithm is listed below:

**Algorithm 4** EXP4($\gamma \in [0,1], T$)

$\mathbf{w}^{(1)} \leftarrow \mathbf{1} \in \mathbb{R}^N$ weights over experts
**for** $t = 1, \cdots, T$ **do**
  RECEIVE($X^{(t)} \in \mathbb{R}^{N \times K}$) advice from N experts
  $\mathbf{q}^{(t)} = \frac{\mathbf{w}^{(t)}}{\|\mathbf{w}^{(t)}\|}$ $\mathbf{X}^{(t)} \in \Delta^K$ probability over actions
  $\mathrm{k}^{(t)} \sim$ MULTINOMIAL($\mathbf{q}^{(t)}$) draw action
  RECEIVE($r^{(t)}$) get reward
  $\hat{\mathbf{r}}^{(t)} = \frac{r^{(t)}}{q_k^{(t)}} \mathbb{I}[k = k^{(t)}] \in \mathbb{I}^K$ reward over all arms
  $\mathbf{g}^{(t)} = \mathbf{X}^{(t)} \cdot \hat{\mathbf{r}}^{(t)} \in \mathbb{R}^N$ per expert reward
  $w_k^{(t+1)} = w_k^{(t)} e^{\gamma g_n^{(t)}}$ $\forall n$
**end for**

The regret bound of EXP4 is

$$R_{EXP4} \leq \sqrt{KT \log N}.$$

As for the connection between the problems EXP3/EXP4, the EXP3 belongs to Multi-Armed Bandit algorithm, while EXP4 belongs to the Contextual Multi-Armed Bandit algorithm. EXP4 can deal with the problems with contextual information and in this way, it can solve better than EXP3. Both of them provide one-shot and evaluative feedback. However, we can know the results of every arm in MAB but we can't implement that in the C-MAB since the reward function was only partial observed. So EXP4 got a sampled feedback, while EXP3 don't.

# 2 Summary

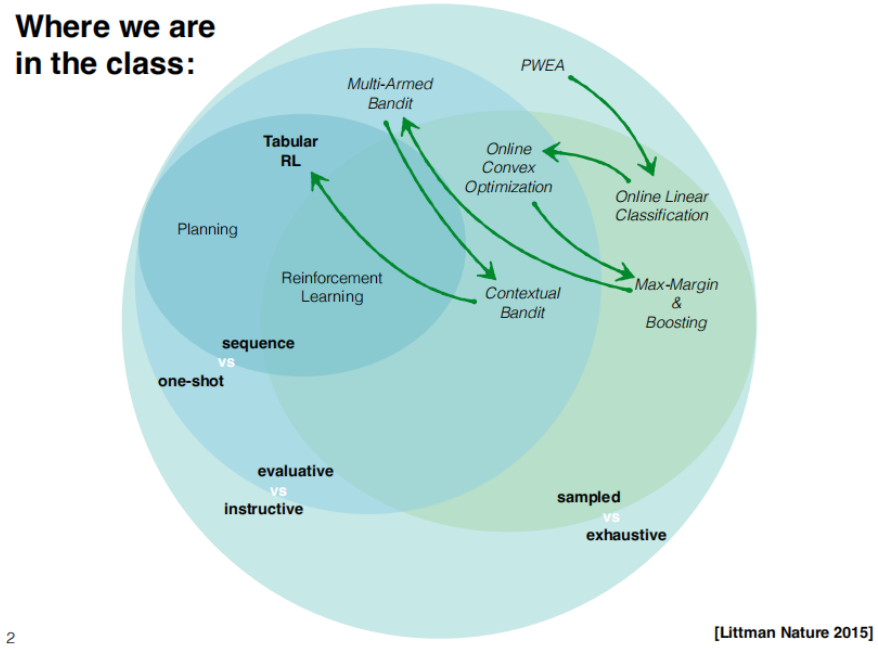


Figure 1: The knowledge that we have covered up to now.

## 2.1 Comparison One-shot learning vs Sequential learning

Up to now, we have studied several online learning algorithms to figure out different problems. However, the feedback of them are all 'one-shot', which means that the samples are identically and independently distributed and will not affect the future state or action in the environment. However, in this course, we will move on to learn the **sequential** learning problems which all samples are drawn through 'interactions' [1]. In this setting, each action taken in this step will affect the state in the future. Though the learning map is still for every state and action pairs, we only acquire feedback/reward after the whole sequence. We will firstly compare the supervised learning and reinforcement learning and then list several types of algorithms that we have learned up to now.

### 2.1.1 Supervised Learning

The assumption is that all the samples are identically and independently drawn.

We define $x$ as a state and $a$ as the action and $\mathcal{D}(x, a)$ is the distribution of state-action pairs, such that we can draw action-state pairs such as $x, a \sim \mathcal{D}(x, a)$. Now using these definitions we can define a stateset as a set of these state-action pairs drawn from the defined distribution, that is $\{(x_1, a_1), (x_2, a_2), \ldots, (x_N, a_N)\}$.

The objective of the supervised learning is to learn a mapping $f : x \to a$, which map the state $a_i$ to the paired action $x_i$. Note that since the previous action will not affect the future state, the

supervised learning will generate **one-shot** feedback. Besides, it is an **instructive** feedback since we know whether the feedback is right or wrong based on the label.

### 2.1.2 Reinforcement Learning

The assumption of the reinforcement learning is that samples are correlated with each other [3]. They are drawn from through the interaction in the environment.

Formally, we define $x$ as a state, $a$ as the action and the action taken in the previous step will affect the state in the next step. Such an example of many $(x, a)$ state-action pairs in different step forms a 'trajectory' $\zeta$ (data points). From a specific trajectory, we can acquire a reward, which those trajectory and reward are in some distribution $\zeta, R \sim \mathcal{D}(\zeta, R)$.

The final dataset is composed of $N$ trajectory-reward pairs, which can be formed as below:

$$\{(\zeta_1, R_1), (\zeta_2, R_2), \ldots, (\zeta_N, R_N)\}$$

where $\zeta_i = \{(x_1, a_1), (x_2, a_2), \ldots, (x_T, a_T)\}$. The goal of this algorithm is to learn a mapping from state x to action a

$$\pi : x \to a$$

under the condition that giving a trajectory $\zeta$ and a reward $R$.

Since we don't know the entire state space, the trajectories are **sampled**. Besides, we can only evaluate how good or bad that the action is but we don't know if it is a correct answer, so RI is a **evaluative** problem. Lastly, since the action can affect the future, reinforcement learning is a **sequence** problem.

### 2.1.3 Review of Learning Problem

| Problem | Sampled | Evaluative | Sequential |
|---|---|---|---|
| PWEA | ✗ | ✗ | ✗ |
| OLC | ✓ | ▲ | ✗ |
| MAB | ✗ | ✓ | ✗ |
| C-MAB | ✓ | ✓ | ✗ |
| RL | ✓ | ✓ | ✓ |
| IL | ✓ | ✓ | ▲ |

Table 1: The properties of several algorithms that we have learnt so far.

The above table contains several algorithms that we have learnt so far. (1).For PWEA, we can obtain all the prediction of the experts, so it is exhausted. Since we know the whether the experts' predictions are right or wrong, the algorithm is instructive. And all parameters (expert advice, observations) could be updated at every step (2).For OLC, the observations are sampled since we have no access to know all the possible space. This algorithm can be both instructive and evaluative which depends on the form of loss that we adopt. If we adopt the zero-one loss, it is instructive since we can know whether it is correct. However, if we adopt the hedge loss, we can only evaluate how good/bad the algorithm is. (3).For MAB and C-MAB, we can know the results of every arm

in MAB but we can't implement that in the C-MAB since the reward function was only partial observed. We could only update one (arm) parameter at a time. All these four algorithms are given one-shot feedback since feedback is obtained each step.

### 2.2 Concept Definitions of Markov Decision Process

A Markov decision process is a discrete-time stochastic control process [2], which is a typical sequence decision making algorithm. Shown in Figure 2, it is a simple example of the Grid world which the whole space can be seen as a concrete space. The position that the robot stands can be seen as a state $S$, which is also an initial state. The initial position that the robot stands is a initial state $s_0$. In each step, the robot can select an action $a_0$ from the set of $\{up, down, left, right\}$. Finally, a sequence of states and actions form as a trajectory $\zeta = \{s_0, a_0, s_1, a_1 \ldots, s_T, a_T\}$. After the whole steps, we can acquire the feedback of the reward by adding each reward along the routes together. In conclusion, the whole process can be seen as Markov Decision Process.

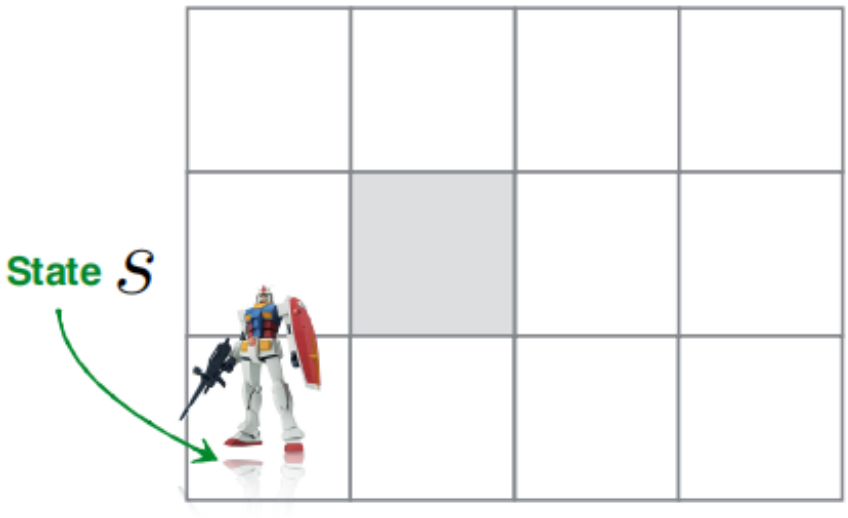


Figure 2: A simple Grid world example.

From it, the probability of a state-action trajectory is defined as:

$$p\left(s_0, a_0, \cdots, s_T, a_T\right)$$

While the reward value is a scalar value for one trajectory is defined as:

$$r\left(s_0, a_0, \cdots, s_T, a_T\right)$$

Before moving forward to the setup of the Markov Decision Process, some concept definitions are

needed.

$s \in S$ State
$a \in A_s$ Action
$p(s^{'}|s,a)$ State Transition Dynamic
$r(s^{'}|s,a)$ Reward function
$p_0(s)$ State prior
$\pi(a|s)$ Policy
$\gamma$ Discount factor

- **State.** The state defines the current situation of the agent (for eg: it can be the exact position of the Robot in the house, or the alignment of its two legs, or its current posture; it depends on how you address the problem) at some time step. There can be finite or infinite states based on the situation. The example mentioned above is a discrete one, while if we define the state as the amount of oil in a vehicle, it would be continuous.

- **Action.** The choice that the agent makes at the current time step with the environment (for eg: it can move its right or left leg, or raise its arm, or lift an object, turn right or left, etc.). We know the set of actions (decisions) that the agent can perform in advance. In this example, it can be move $\{up, down, left, right\}$. Generally, it can allow us to change the current state or the environment.

- **State-Transition Dynamics.** In the real environment, there exists some uncertainty. So we are not for sure what the outcome of it even if we know the state $s$ and the action $a$. And $p(s^{'}|s,a)$ is probability to describe that, which is a the transition dynamics of the environment, and describes the probability that action $a$ in state $s$ will lead to $s^{'}$,

- **Reward Function.** $r(s^{'}|s,a)$ is the immediate reward received from the environment, which help us model the intention of the agents that they want to achieve. By maximizing its reward, we can get closer to what the agent want. For example, we own the amount money of $x$ and we want to earn the number of money $y$. Therefore, the reward function can be the inversely proportional to the difference from the target money.

- **State prior.** $p_0(s)$ is the initial state probability, which describes the probability of the environment or the agent being initialized to state $s_0$.

- **Policy.** As a probability distribution, a policy is the thought process behind picking an action. In practice, it is a probability distribution assigned to the set of actions. Highly rewarding actions will have a high probability and vice versa. If the agent follow the policy $\pi$, it will perform the $s$ with the probability $\pi(a|s)$ under the condition that the current state is $a$.

- **Discount factor.** The variable $\gamma \in [0, 1]$ is the discount factor. The intuition behind using a discount is that there is no certainty about the future rewards; i.e., as important it is to consider the future rewards to increase the Return, it is also equally important to limit the contribution of the future rewards to the Return (Since you can't be 100% sure of the future). When we look at an agents behavior over a series of time steps, we generally like to sum all of the rewards at every time step to get the return of the trajectory. In this sum, we can choose

to multiply future rewards by a power of $\gamma$ to control the value of weights that we put in the future rewards.

## 2.3 Factorization and Procedure of MDP

As a sequential decision making algorithm, we can factorize the MDP to a temporal sequence of variables. As defined above, a trajectory as a sequence of states and actions $\zeta = \{s_0, a_0, s_1, a_1, ..., s_T, a_T\}$. The joint distribution of generating a trajectory $p(s_0, a_0, s_1, a_1, ..., s_T, a_T)$ and a reward function $r(s_0, a_0, s_1, a_1, ..., s_T, a_T)$, which is a scalar value for one trajectory. In this way, the state $s_t$ can capture all the relevant information and we can factorize the MDP after knowing the transition dynamics $p(s_{t+1}|s_t, a_t)$, a policy $\pi(a_t|s_t)$, and the Markov property. The equation is as below:

$$p(s_0, a_0, ..., s_T, a_T) = p_0(s_0)\, \Pi_{t=0}^{T-1}\, p(s_{t+1}|s_t, a_t)\pi(a_t|s_t)$$

We will only acquire one feedback based on the whole sequence. And there are many possible factorizations of the return which can be defined as follows:

$$\begin{aligned} r(s_0, a_0, s_1, a_1, ..., s_T, a_T) =& r(s_0, a_0, s_1) + r(s_1, a_1, s_2) + ... && \text{or} \\ & r(s_0, a_0) + r(s_1, a_1) + ... && \text{or} \\ & r(s_0) + r(s_1) + ... \end{aligned}$$

## 2.4 Mathematic of the MDP

### 2.4.1 Value Function

As the model of reinforcement learning needs to model the future, we should also know how to evaluate the performance of the modeling, which is what the value function does. For a trajectory, the value function gives a feedback. And there are two major forms of value function: state value function $V^\pi(s)$ which gives a feedback of a trajectory starting in state $s$, and a action value function $Q^\pi(s, a)$ which gives a feedback of a trajectory starting in state $s$ and performing action $a$.

**State Value Function** The state value function can be written out as:

$$V^\pi(s) = \mathbb{E}_p \left[ r_0 + r_1 + r_2 + \cdots \mid s_0 = s \right],$$

in this equation, $\pi$ is the followed policy; $s$ is the starting state; $p$ is the distribution with the value of:

$$p(s_0, a_0, s_1, a_1, \cdots) = p_0(s_0)p(s1|s_0, a_0)p(a_0|s_0)p(s_2|s_1, a_1)p(a_1|s_1)\cdots$$

The value function can be defined with respect to different time horizons. The Infinite horizon return is defined as follows:

$$V^\pi(s) = \mathbb{E}_p \left[ r_0 + r_1 + r_2 + \cdots \mid s_0 = s \right],$$

while the finite horizon return is defined as follows:

$$V^\pi(s) = \mathbb{E}_p \left[ r_0 + r_1 + r_2 + \cdots r_T \mid s_0 = s \right],$$

and the infinite horizon discounted return is defined as follows:

$$V^{\pi}(s) = \mathbb{E}_p \left[ \gamma^0 r_0 + \gamma^1 r_1 + \gamma^2 r_2 + \cdots \mid s_0 = s \right].$$

The infinite horizon discounted return is reasonable since we need to focus more on the recent state and care less about the future ones. In that way, we should model them by multiplying a discount parameter.

**State-Action Value Function**

The state-action value function means that the total feedback of a trajectory starting in the state $s$ and corresponding action $a$:

$$Q^{\pi}(s,a) = \mathbb{E}_p \left[ \gamma^0 r\left(s_0\right) + \gamma^1 r\left(s_1\right) + \gamma^2 r\left(s_2\right) + \cdots \mid s_0 = s, a_0 = a \right].$$

From the equations of $V^{\pi}(s)$ and $Q^{\pi}(s,a)$, we can find the connection between them below:

$$V^{\pi}(s) = \sum_a \pi(a \mid s) Q^{\pi}(s,a)$$

The process have been listed below:

$V^{\pi}(s) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s\right]$ A refined version of the $V^{\pi}(s)$ equation.

$= \sum_{s_{1:\infty}, a_{0:\infty}} p\left(s_{1:\infty}, a_{0:\infty}\right) \left[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s\right]$ It is a mean of all the possible state in the trajectory, so for every state and corresponding action, we should multiply the probability of it with reward function.Finally sum them together.

$= \sum_{s_{1:\infty}, a_0:\infty} \pi\left(a_0 \mid s_0 = s\right) p\left(s_{1:\infty}, a_{1:\infty}\right) \left[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s\right]$ Split the first probability since it equals to the probability from the state $s$ to take the action $a_0$.

$= \sum_a \pi\left(a_0 = a \mid s_0 = s\right) \sum_{s_{1:\infty}, a_{1:\infty}} p\left(s_{1:\infty}, a_{1:\infty}\right) \left[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s, a_0 = a\right]$ Split the policy from the whole sum. And sum all the possible actions together.

$= \sum_a \pi\left(a_0 = a \mid s_0 = s\right) \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s, a_0 = a\right]$ The latter part equals to starting from a state $s$ and an action $a$.

$= \sum_a \pi(a \mid s) Q^{\pi}(s,a)$

### 2.4.2 Bellman Equation

It describe the recurrent relationship between value functions under a policy. We can find its definition in the Figure below:

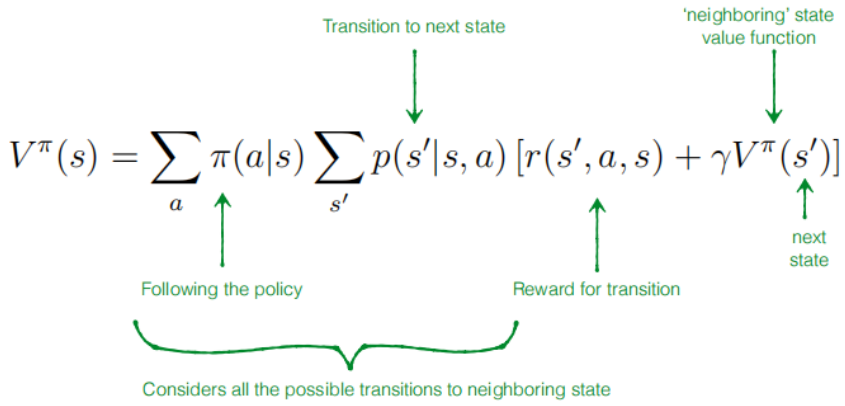


Figure 3: The Bellman Equation.

The process can be explained as follows. For the first sum, it sums for all the possibility of all action $a$ that for state $s$ can take. The second sum is the possibility to reach the state of $s^{'}$ under the condition of state $s$ and action $a$. The last part is the reward when transferring to $s^{'}$ and corresponding reward starting from $s^{'}$.

The process of obtaining the equations of $V^\pi(s_0)$ and $Q^\pi(s_0, a_0)$ have been listed below:

$V^\pi(s_0) = \mathbb{E}\left[\gamma^0 r_0 + \gamma^1 r_1 + \gamma^2 r_2 + \cdots \mid s_0\right]$ Original equation.

$= \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0\right]$ A refined version of the $V^\pi(s)$ equation.

$= \mathbb{E}\left[r_0 + \sum_{t=1}^{\infty} \gamma^t r_t \mid s_0\right]$ Extract $r_0$ from the equation.

$= \sum_{a_{0:\infty}} \sum_{s_{1:\infty}} p\left(s_{1:\infty}, a_{0:\infty}\right)\left[r_0 + \sum_{t=1}^{\infty} \gamma^t r_t \mid s_0\right]$ In this line, we need to separately add different actions $a$ and states $s$ together. For each state, we should sum the probability that for all possible initial actions. And then, we should sum up all the probabilities that can reach different neighbour states.

$= \sum_{a_0} \sum_{s_1} p(s_1 \mid s_0, a_0) \pi\left(a_0 \mid s_0\right)\{r_0 + \sum_{a_{1:\infty}} \sum_{s_{2:\infty}} p\left(s_{2:\infty}, a_{1:\infty}\right)\left[\sum_{t=1}^{\infty} \gamma^t r_t \mid s_1\right]\}$Keep action $a_0$ and state $s_1$ outside and move the sum to infinity inside the bracket.

$V^{\pi}(s_0) = \sum_{a_0}\sum_{s_1} p(s_1 \mid s_0, a_0)\pi\left(a_0 \mid s_0\right)\{r_0 + \mathbb{E}\left[\sum_{t=1}^{\infty}\gamma^t r_t \mid s_1\right]\}$The latter part equals to starting from a state $s_1$.

$= \sum_{a_0}\pi\left(a_0 \mid s_0\right)\sum_{s_1} p(s_1 \mid s_0, a_0)\{r_0 + \gamma V^{\pi}(s_1)\}$

$Q^{\pi}(s_0, a_0) = \mathbb{E}\left[\gamma^0 r_0 + \gamma^1 r_1 + \gamma^2 r_2 + \cdots \mid s_0, a_0\right]$ Original equation.

$= \mathbb{E}\left[\sum_{t=0}^{\infty}\gamma^t r_t \mid s_0, a_0\right]$ A refined version of the $V^{\pi}(s_0, a_0)$ equation.

$= \mathbb{E}\left[r_0 + \sum_{t=1}^{\infty}\gamma^t r_t \mid s_0, a_0\right]$ Extract $r_0$ from the equation.

$= \sum_{s_{1:\infty}} p\left(s_{1:\infty}, a_{0:\infty}\right)\left[r_0 + \sum_{t=1}^{\infty}\gamma^t r_t \mid s_0, a_0\right]$ Since the initial action has been ensured, we only sum all the possible probabilities that reach its neighbours.

$= \sum_{s_1} p\left(s_1 \mid s_0, a_0\right)\{r_0 + \mathbb{E}\left[\sum_{t=1}^{\infty}\gamma^t r_t \mid s_1\right]\}$This time, we transfer the state $s_0$ to its neighbour $s_1$ by leveraging action $a_0$. Therefore, the inside equation become the mean of starting from state $s_1$.

$= \sum_{s_1} p\left(s_1 \mid s_0, a_0\right)\{r_0 + \sum_{a_1}\pi(a_1 \mid s_1)\mathbb{E}\left[\sum_{t=1}^{\infty}\gamma^t r_t \mid s_1, a_1\right]\}$We take the second action as $a_1$ for every neighbour and then sum the probabilities of different policies of taking different actions together.

$= \sum_{s_1} p\left(s_1 \mid s_0, a_0\right)\{r_0 + \sum_{a_1}\pi(a_1 \mid s_1)Q^{\pi}(s_0, a_0)\}$Final equation.

The Bellman Equations define a (recursive) relationship between value functions and are also used to derive the Bellman optimality equations. Besides, it gives us a useful constraint for optimization.

### 2.4.3 Bellman Optimality Equations

Recurrent relationship between value functions under an optimal policy. For all the appeared policy, we need to select the policy with the more reward. Therefore, we can acquire the optimal equations based on $V^{\pi}(s)$ and $Q^{\pi}(s, a)$ are defined as follows:

$$V^{\pi^*}(s) = \max_a \sum_{s'} p(s' \mid s, a)\left[r_t + \gamma V^{\pi^*}(s_{\prime})\right]$$

$$Q^{\pi^*}(s,a) = \sum_{s\prime} p\left(s\prime \mid s,a\right)\left[r(s) + \gamma \max_{a'} Q^{\pi^*}(s^{'},a^{'})\right]$$

The process of obtaining the equation of $V^{\pi^*}(s)$ has been listed below:

$V^{\pi^*}(s) = \sum_a \pi(a \mid s) Q^{\pi^*}(s,a)$Sum the probability of all possible actions for $Q^{\pi^*}(s,a)$ with the initial action as $a$.

$= \max_a Q^{\pi^*}(s,a)$Change the policy into a deterministic one. For the best policy, we need to select the initial action $a$ which can maximize the reward.

$= \max_a \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t \mid s_0 = s, a_0 = a\right]$ Change the $Q$ into a sum version.

$= \max_a \mathbb{E}\left[r_0 + \sum_{t=1}^{\infty} \gamma^t r_t \mid s_0 = s, a_0 = a\right]$ Divide the first reward from the equation.

$= \max_a \sum_{s'} p(s_1 = s^{'} \mid s,a)\left[r_0 + \mathbb{E}\{\sum_{t=1}^{\infty} \gamma^t r_t \mid s_1 = s^{'}\}\right]$ Move one step from state $s$ to state $s^{'}$.

Therefore, the equation inside the bracket should start from the state $s\prime$

$= \max_a \sum_{s'} p(s^{'} \mid s,a)\left[r_t + \gamma V^{\pi^*}(s\prime)\right]$

## References


[1] L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

[2] M. L. Puterman. Markov decision processes. *Handbooks in operations research and management science*, 2:331–434, 1990.

[3] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

# 3 Appendix

## 3.1 Factored MDPs

In the Appendix, we would like to talk about the development and some approaches about the Factored MDPs.

**Challenges of MDPs:** Markov Decision Processes (MDPs) have been used as the basic semantics for optimal planning for decision theoretic agents in stochastic environments. In the MDP framework, the system is modeled via a set of states which evolve stochastically. The main problem with this representation is that, in virtually any real-life domain, the state space is quite large. However, many large MDPs have significant internal structure, and can be modeled compactly if the structure is exploited in the representation.

**Definition of FMDPs:** The Factored MDPs are one approach to representing large, structured MDPs compactly. In this framework, a state is implicitly described by an assignment to some set of state variables. A dynamic Bayesian network (DBN) [T. Dean and K. Kanazawa; "A model for reasoning about persistence and causation"] can then allow a compact representation of the transition model, by exploiting the fact that the transition of a variable often depends only on a small number. The state space of the FMDPs can be specified as a cross-product of sets of state variables $Ei(E = E0E1En)$. A factored formulation also allows for system dynamics to be specified using a more natural and intuitive representation instead of an $SS$ probability matrix per action.

**Two types of structure exploited in FMDPs:** There are two main types of structure that can simultaneously be exploited in factored MDPs: additive and context-specific structure. Additive structure captures the fact that typical large-scale systems can often be decomposed into a combination of locally interacting components. For example, consider the management of a large factory with many production cells. Of course, in the long run, if a cell positioned early in the production line generates faulty parts, then the whole factory may be affected. However, the quality of the parts a cell generates depends directly only on the state of this cell and the quality of the parts it receives from neighboring cells. Such additive structure can also be present in the reward function. For example, the cost of running the factory depends, among other things, on the sum of the costs of maintaining each local cell.

Context-specific structure encodes a different type of locality of influence: Although a part of a large system may, in general, be influenced by the state of every other part of this system, at any given point in time only a small number of parts may influence it directly. In our factory example, a cell responsible for anodization may receive parts directly from any other cell in the factory. However, a work order for a cylindrical part may restrict this dependency only to cells that have a lathe. Thus, in the context of producing cylindrical parts, the quality of the anodized parts depends directly only on the state of cells with a lathe.

**Several approaches for solving the FMDPS** In [C. Boutilier, R. Dearden, and M. Goldszmidt; "Exploiting structure in policy construction"], the authors exploit such a factored state space directly, and reveal reduction in the computation and memory required to compute the optimal solution. The assumption is that the MDP is specified by a set of DBNs, one for each action, although the claim made is that it is amenable to a probabilistic STRIPS specification too. In addition to using the network structure to elicit variable independence, they use decision-tree representations of the conditional probability distributions (CPDs) to further exploit propositional

independence. Next, they construct the structured policy iteration (SPI) algorithm which aggregates states for two distinct reasons: either if the states are assigned the same action by the current strategy, or if states have the same current estimated value. With the aggregation in place, the learning algorithm based on modified policy iteration only computes at the coarser level of these state partitions instead of that of the individual states. The algorithm itself is split into two phases, structured successive approximation and structured policy improvement, mirroring the two phases of classical policy iteration. It is important to note that SPI will see fewer advantages if the optimal strategy cannot be compactly represented by a tree structure, and for the reason that there is still big overhead in finding the state partitions.

In [R. A. Howard; Dynamic Programming and Markov Processes], Algebraic Decision Diagrams (ADDs) replace the decision-tree learning of SPI for the value function and strategy representation. The paper deals with a very large MDP (about 63 million states) and shows that the learned ADD value function representation is considerably more compact than the corresponding learned decision tree in most cases. However, a big disadvantage of using ADDs is that the state variables must be boolean, which makes the modified state space larger than the original.

In order to solve large weakly coupled FMDPs, the state space of the original MDP is divided into regions that comprise sub-MDPs which run concurrently (the original MDP is a cross-product of the sub-MDPs) [N. Meuleau, M. Hauskrecht, K. Kim; "Solving very large weakly coupled Markov decision processes"]. It is assumed that states variables are only associated with a particular task and the numbers of resources that can be allocated to the individual tasks are constrained; these global constraints are what cause the weak coupling between the sub-MDPs. Their approach contains two phases: an offline phase that computes the optimal solutions (value functions) for the individual sub-MDPs and an online phase that uses these local value functions to heuristically guide the search for global resource allocation to the subtasks.

**Approaches for solving the weakly coupled FMDPS** One class of methods for solving weakly coupled FMDPs involves the use of linear value function approximation. In [E. Hansen and Z. Feng; "Dynamic programming for POMDPs using a factored state representation"], the authors present two solution algorithms (based o approximate linear and dynamic programming) that approximate the value functions using a linear combination of basis functions, each basis function only depending on a small subset of the state variables. In [P. Poupart, C. Boutilier, R. Patrascu, and D. Schuurmans; "Piecewise linear value function approximation for factored MDPs"], a general framework is proposed that can select a suitable basis set and modify it based on the solution quality. Further, they use piecewise linear combination of the subtask value functions to approximate the optimal value function for the original MDP. The above approaches to solving FMDPs are classified under decision-theoretic planning in that they need a perfect model (transition and reward) of the FMDP. The work in [T. Degris, O. Sigaud, and P.-H. Wuillemin; "Learning the structure of factored markov decision processes in reinforcement learning problems"] proposes the SDYNA framework that can learn in large FMDPs without initial knowledge of their structure. SDYNA incrementally builds structured representations using incremental decision-tree induction algorithms that learn from the observations made by the agent.