I. Introduction to Reinforcement Learning

1. Introduction

Reinforcement learning (RL) is a science of *decision-making*. We are going to discuss RL from the perspective of machine learning (ML).

What makes RL different from other ML algorithms (e.g. supervised/unsupervised learning)? [1.1]

(a) There is no supervisor, only a *reward* signal.

(b) Feedback is delayed, not instantaneous.

(c) Time really matters (i.e. sequential, non iid data is supplied).

(d) *Agent*'s action affect the subsequent data it receives.

Example 1.1. Examples of RL

(a) Fly stunt manoeuvering in a helicopter.

(b) Playing Backgammon.

(c) Managing an investment portfolio.

(d) Controlling a power station.

(e) Making a humanoid robot walk.

(f) Playing Atari games.

2. The RL Problem

Def'n 1.1. **Reward**
A ***reward*** R_t at ***step*** t is a scalar feedback signal indicating how well agent is doing at step t.

The job of an RL agent is to maximize cumulative reward. RL is based on the following hypothesis.

Statement 1.1. Reward Hypothesis
Goals can be described by maximization of expected cumulative reward.

As a consequence of the above hypothesis, we have:

(a) Actions may have long term sequence.

(b) Reward may be delayed. *It may be better to sacrifice immediate reward to gain more long-term reward.*

For instance, in a financial investment, it may take months to mature.

We can describe RL by using the relationship between agent and environment: at each step t, the agent

(a) executes action A_t (towards the environment); and

(b) receives observation O_t and scalar reward R_t from the environment.

We repeat this over and over to train our agent.

Def'n 1.2. **History**
The ***history*** H_t at step t is the sequence of observations, actions, rewards up to time t:

$$H_t = \left((A_i, O_i, R_i)\right)_{i=1}^{t} = \left(A_1, O_1, R_1\right), \ldots, \left(A_t, O_t, R_t\right).$$

What happens next depends on the history. The agent select actions and the environment selects observations and rewards. But history is often too long, so instead we use less amount of information to determine what happens next.

Def'n 1.3. **State**
The ***state*** S_t at step t is the information used to determine what happens at step $t+1$.

There are three parts of state.

Def'n 1.4. **Environmental State**
The ***environmental state*** S_t^e at step t is the environment's private representation that spits out next observation and reward.

The environment state is not usuall visible to the agent. Even if it is visible, it may contain irrelevant information.

Def'n 1.5. **Agent State**
The ***agent state*** S_t^a is the agent's internal representation the agent uses to pick the next action. Formally, it is a function of the history:

$$S_t = f(H_t)$$

for some function f.

When we speak of state, we shall always mean agent state.

Def'n 1.6. **Markov** Sequence of State
We say the sequence of states $(S_t)_{t=1}$ is ***Markov*** if

$$\mathbb{P}(S_{t+1}|S_1, \ldots, S_t) = \mathbb{P}(S_{t+1}|S_t)$$

for all t.

In words, for a Markov process,

the future independent of the past given the present. [1.2]

In other words, once S_t is known, the history may be thrown away (i.e. the state S_t is a sufficient statistic of the future). By definition, the environment state S_t^e and the history H_t are Markov.

Def'n 1.7. **Fully Observable**, **Partially Observable** Environment
A ***fully observable*** environment is when the agent directly observes the environmental stae. That is,

$$S_t^a = S_t^e.$$

Otherwise, we say that the environment is ***partially observable***.

In a fully observable environment, $(S_t)_{t=1}$ is a *Markov decision process* (MDP), to which we will come back shortly.

3. Inside an RL Agent

An RL agent may include one or more of these components.

Def'n 1.8. **Policy**, **Value Function**, **Model** of an RL Agent
The ***policy*** π of an RL agent is the agent's behaviour function. A ***deterministic*** policy is a deterministic function from state s to action a:

$$a = \pi(s)$$

for all state s. A ***stochastic policy*** has the form of

$$\pi(a|s) = \mathbb{P}(A = a|S = s).$$

We desire π such that we can get back as much rewards back as possible from a.

The ***value function*** is the measure of how good is each state or action (i.e. a prediction of future reward). Formally,

$$v_\pi(s) = \mathbb{E}_\pi\left(R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \cdots | S_t = s\right)$$

for some $\gamma \in [0, 1]$.[1]

The ***model*** is the agent's representation of the environment that predicts what the environment will do next. There are two parts of a model. A ***transition*** model $\mathcal{P}$ predicts the next state:

$$\mathcal{P}^a_{ss'} = \mathbb{P}\left(S' = s' | S = s, A = a\right).$$

A ***reward*** model $\mathcal{R}$ predicts the next reward:

$$\mathcal{R}^a_s = \mathbb{E}\left(R | S = s, A = a\right).$$

[1] The parameter γ *says* that we care about immediate rewards more than future rewards (unless future rewards are much larger).

Def'n 1.9. **Value-based**, **Policy-based**, **Actor Critic** RL Agent
We say an RL agent is

(a) ***value-based*** if it has a value function but no policy;[1]

(b) ***policy-based*** if it has a policy but no value function; and

(c) ***actor critic*** if it has both.

[1] A value-based RL agent *reads off* value function *greedily* to come up with an implicit policy.

Def'n 1.10. **Model-free**, **Model-based** RL Agent
We say an RL agent is

(a) ***model-free*** if it has no model; and

(b) ***model-based*** if it has one.

4. Problems within RL

There are two fundamental problems in sequential decision making.

	RL	Planning
Environment	initially unknown	a model is known
Agent	interacts with the environment to improves its policy	performs computation with its model without external interactions to improve policy

In short,

RL is like a trial-and-error learning. [1.3]

The agent should discover a good policy from its experiences of the environment, without losing too much reward along the way. In doing so, *exploration* finds more information about the environment and *exploitation* exploits known information to maximize reward. We have to balance exploration and exploitation to get the best result.

In RL, we would like to *optimize the future* (i.e. find the best policy). This is called *control*. It turns out we have to make *predictions* (i.e. given a policy, evaluate the future) to do so.

II. Markov Decision Processes

As mentioned in Section 1, *Markov decision processes* (MDPs) formally describe an environment for RL when the environment is *fully observable*. That is, the current state completely characterizes the process. The nice thing about MDPs is that most RL problems can be formalized in to MDPs. For instance,

(a) optimal constrol primarily deals with continuous MDPs;

(b) partially observable problems can be converted in to MDPs; and

(c) bandits are MDPs with one state.

1. Markov Processes

Let us start by recalling the definition of *Markov property*.

Recall 2.1. **Markov Process**

A ***stochastic process*** is a sequence $(X_t)_t$ of random variables.

We say a stochastic process $(S_t)_t$ is ***Markov*** if

$$\mathbb{P}\left(S_{t+1}|S_t,\ldots,S_1\right) = \mathbb{P}\left(S_{t+1}|S_t\right).$$

For a Markov state s and successor state s', the ***state transition probability***, denoted as $\mathcal{P}_{ss'}$, is defined by

$$\mathcal{P}_{ss'} = \mathbb{P}\left(S_{t+1} = s'|S_t = s\right).$$

The ***state transition matrix*** $\mathcal{P}$ defines transition probabilities from all states s to all successor states s',

$$\mathcal{P} = \begin{bmatrix} \mathcal{P}_{11} & \cdots & \mathcal{P}_{1n} \\ \vdots & \ddots & \vdots \\ \mathcal{P}_{n1} & \cdots & \mathcal{P}_{nn} \end{bmatrix}.$$

This means we can characterize a Markov process $(S_t)_t$ by a pair $(\mathcal{S},\mathcal{P})$, where $\mathcal{S}$ is the (finite) set ofpossible states and $\mathcal{P}$ is the state transition matrix. For this reason, we shall also call $(\mathcal{S},\mathcal{P})$ a ***Markov process***.

Def'n 2.2. **Markov Reward Process**

A ***Markov reward process*** (***MRP***) is a tuple $(\mathcal{S},\mathcal{P},\mathcal{R},\gamma)$ such that

(a) $(\mathcal{S},\mathcal{P})$ is a Markov process;

(b) $\mathcal{R}$ is a ***reward function*** with

$$\mathcal{R}_s = \mathbb{E}\left(R_{t+1}|S_t = s\right)$$

for all $s \in \mathcal{S}$;[1] and

(c) $\gamma \in [0,1]$, called the ***discount factor***.

[1] R_t represents the ***reward*** at step t.

Def'n 2.3. **Return**

The ***return*** from step t, denoted as G_t, is the total discounted reward from t:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots.$$

According to the definition of G_t, the value of receiving reward R after $k+1$ steps is $\gamma^k R$, so G_t values immediate reward above delayed reward. Specifically, $\gamma \approx 0$ leads to ***myopic*** evaluation whereas $\gamma \approx 1$ leads to ***far-sighted*** evaluation.

One reason that we use a discount factor is because of the *uncertainty* in the future. To put this in another way, we *do not* have a perfect model of the environment.

Another reason is for mathematical convenience: by using discount rewards, we can avoid infinte returns (e.g. in cyclic Markov processes).

Def'n 2.4. **State Value Function**
The ***state value function***, denoted as $v(s)$, of an MRP is the expectation of G_t starting from state s:

$$v(s) = \mathbb{E}\left(G_t | S_t = s\right)$$

for all state s.

By defintion, we have

$$\begin{aligned} v(s) &= \mathbb{E}\left(G_t | S_t = s\right) \\ &= \mathbb{E}\left(R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots | S_t = s\right) \\ &= \mathbb{E}\left(R_{t+1} + \gamma\left(R_{t+2} + \gamma R_{t+3} + \cdots\right) | S_t = s\right) \\ &= \mathbb{E}\left(R_{t+1} + \gamma G_{t+1} | S_t = s\right) \\ &= \mathbb{E}\left(R_{t+1} + \gamma v\left(S_{t+1}\right) | S_t = s\right). \end{aligned}$$

That is, the value function $v(s)$ has two parts: immediate reward R_{t+1} and discount value of successor state $\gamma v\left(S_{t+1}\right)$. The equation

$$v(s) = \mathbb{E}\left(R_{t+1} + \gamma v\left(S_{t+1}\right) | S_t = s\right) \tag{2.1}$$

is called the *Bellman equation*. This also can be expressed concisely using matrices:

$$v = \mathcal{R} + \gamma \mathcal{P} v, \tag{2.2}$$

where v is a column vector with on entry per state. That is,

$$\begin{bmatrix} v(1) \\ \vdots \\ v(n) \end{bmatrix} = \begin{bmatrix} \mathcal{R}_1 \\ \vdots \\ \mathcal{R}_n \end{bmatrix} + \gamma \begin{bmatrix} \mathcal{P}_{11} & \cdots & \mathcal{P}_{1n} \\ \vdots & \ddots & \vdots \\ \mathcal{P}_{n1} & \cdots & \mathcal{P}_{nn} \end{bmatrix} \begin{bmatrix} v(1) \\ \vdots \\ v(n) \end{bmatrix}. \tag{2.3}$$

[2.3] is a linear equation, which can be solved directly:

$$v = \left(I_n - \gamma \mathcal{P}\right)^{-1} \mathcal{R}. \tag{2.4}$$

Evaluating [2.4] has time complexity $O\left(n^3\right)$, so direct solution is only feasible for small MRPs. Instead, we will shortly look into many iterative (approximation) methods for larger MRPs, such as

(a) dynamic programming;

(b) Monte-Carlo evaluation; and

(c) temporal-difference learning.

Def'n 2.5. **Markov Decision Process**
A ***Markov decision process*** (***MDP***) is a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$ such that

(a) $(\mathcal{S}, \mathcal{P}, \mathcal{R}, \gamma)$ is a MRP with additional properties

$$\mathcal{P}^a_{ss'} = \mathbb{P}\left(S_{t+1} = s' | S_t = s, A_t = a\right)$$

and

$$\mathcal{R}^a_s = \mathbb{E}\left(R_{t+1} | S_t = s, A_t = a\right)$$

for all states s, s' and action a; and

(b) $\mathcal{A}$ is a (finite) set of actions.

In short, a MDP is a MRP with decisions.

Def'n 2.6. **Policy**
A (stochastic) ***policy*** π is a distribution over actions given states,

$$\pi\left(a|s\right) = \mathbb{P}\left(A_t = a | S_t = s\right)$$

for all $a \in \mathcal{A}, s \in \mathcal{S}$.

A policy fully characterizes the behavior of an agent. In an MDP, policies depend on the current state but not the whole history. In particular, policies are *stationary* (i.e. time-independent). That is, given any $t, t' > 0$,

$$\mathbb{P}\left(A_{t'} = a | S_{t'} = s\right) = \mathbb{P}\left(A_t = a | S_t = s\right).$$

Suppose that an MDP $\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$ and a policy π are given. Then the state sequence $(S_t)_t$ is a Markov process $(\mathcal{S}, \mathcal{P}^\pi)$ and the state and reward sequence $((S_t, R_t))_t$ is a MRP $(\mathcal{S}, \mathcal{P}^\pi, \mathcal{R}^\pi, \gamma)$, where

$$\mathcal{P}^\pi_{s,s'} = \sum_{a \in \mathcal{A}} \pi\left(a|s\right) \mathcal{P}^a_{s,s'}$$

and

$$\mathcal{R}^\pi_s = \sum_{a \in \mathcal{A}} \pi\left(a|s\right) \mathcal{R}^a_s.$$

This leads to the following two definitions.

Def'n 2.7. **State-value Function**, **Action-value Function** of an MDP
The ***state-value function*** $v_\pi\left(s\right)$ of an MDP is the expectation of return starting from state s and then following policy π:

$$v_\pi\left(s\right) = \mathbb{E}_\pi\left(G_t | S_t = s\right).$$

The ***action-value function*** $q_\pi\left(s, a\right)$ is the expected return starting from state s, taking action a, and ten following policy π:

$$q_\pi\left(s, a\right) = \mathbb{E}_\pi\left(G_t | S_t = s, A_t = a\right).$$

Similar to [2.1], we can decompose the state-value function $v_\pi\left(s\right)$ and the action-value function $q_\pi\left(s, a\right)$ into two parts:

$$v_\pi\left(s\right) = \mathbb{E}_\pi\left(R_{t+1} + \gamma v_\pi\left(S_{t+1}\right) | S_t = s\right) \qquad [2.5]$$

and

$$q_\pi(s,a) = \mathbb{E}_\pi\left(R_{t+1} + \gamma q_\pi\left(S_{t+1}, A_{t+1}\right) | S_t = s, A_t = a\right). \quad [2.6]$$

Writing [2.5] in matrix notation gives

$$v_\pi = \mathcal{R}^\pi + \gamma \mathcal{P}^\pi v_\pi \quad [2.7]$$

with an explicit solution

$$v_\pi = \left(I_n - \gamma \mathcal{P}^\pi\right)^{-1} \mathcal{R}^\pi. \quad [2.8]$$

Def'n 2.8. **Optimal State-value Function**, **Optimal Action-value Function** of an MDP

The ***optimal state-value function*** $v_*(s)$ is the maximum state-value function over all policies:

$$v_*(s) = \max_\pi v_\pi(s).$$

The ***optimal action-value function*** $q_*(s,a)$ is the maximum action-value function over all policies:

$$q_*(s,a) = \max_\pi q_\pi(s,a).$$

We can define the following partial order on policies that captures the idea of *a policy better than another*: we write

$$\pi \geq \pi'$$

if for every $s \in \mathcal{S}$, $v_\pi(s) \geq v_{\pi'}(s)$. The following theorem concerns this partial order.

Theorem 2.1.

For any MDP,

(a) there is an optimal policy π_* such that $\pi_* \geq \pi$ for all policy π; and

(b) any optimal policy π_* achieve the optimal state-vaue function,

$$v_{\pi_*} = v_*,$$

and the optimal action-value function,

$$q_{\pi_*} = q_*.$$

The optimal value functions are recursively related by the Bellman optimality equations:

$$v_*(s) = \max_a q_*(s,a) \quad [2.9]$$

and

$$q_*(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*\left(s'\right). \quad [2.10]$$

This means

$$v_*(s) = \max_a \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*\left(s'\right) \quad [2.11]$$

and

$$q_*(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \max_{a'} q_*\left(s', a'\right). \quad [2.12]$$

[2.9] and [2.10] are non-linear, and they do not have no closed form solution in general. Instead, we use iterative approximation methods.

III. Planning by Dynamic Programming

Starting from this section, we will focus on discussing *solution methods* to MDPs.

1. Introduction

What is *dynamic programming*?

The word *dynamic* stands for *sequential or temporal component* to the problem and *programming* stands for *optimizing a program* (e.g. optimizing a policy for RL).

Dynamic Programming

Given a complex problem,

(a) break it down into subproblems;

(b) solve the subproblems; and

(c) combine solutions to subproblems.

To apply dynamic programming, the program of concern must have two properties:

(a) optimal substructure on where *principle of optimality* applies; and

(b) overlapping subproblems – the subproblems occur many times; we gain something from breaking down the problem into subproblems and by solving those subproblems we solve things more efficiently than attacking the overall problem.

Statement 3.1. Principle of Optimality

The subsolutions of an optimal solution of the problem are optimal solutions for the corresponding subproblems.

MDPs satisfy both properties. In fact, Bellman equation gives recursive decomposition which gives the properties we are after and value function stores and reuses solutions. The value function of a state tells us the optimal solution from that state onwards. When we apply dynamic programming (at least on the theoretical level), we assume full knowledge of the MDP, and use it for *planning*. For prediction:

$$\begin{aligned}&\text{input} - \text{MDP } (\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma) \text{ and policy } \pi,\\ &\text{output} - \text{value function } v_\pi.\end{aligned}$$

For control:

$$\begin{aligned}&\text{input} - \text{MDP } (\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma),\\ &\text{output} - \text{optimal value function } v_* \text{ and optimal policy } \pi_*.\end{aligned}$$

Before we discuss applying dynamic programming on MDPs in depth, here are some applications of dynamic programming in other areas.

Example 3.1. Applications of Dynamic Programming

(a) Scheduling algorithms.

(b) String algorithms (e.g. sequence alignment).

(c) Graph algorithms (e.g. shortest path).

(d) Graphical methods (e.g. Viterbi algorithm).

(e) Bioinformatics (e.g. lattice models).

2. Policy Evaluation

We are going to consider the following problem:

$$\begin{aligned} &\text{given : a policy } \pi \\ &\text{find : iterative application of Bellman expectation backup.} \end{aligned} \tag{3.1}$$

We will start from an arbitrary *value function* v_1 and update it over and over until we *converge* to a true value function v_π:

$$v_1 \to \cdots \to v_\pi.$$

Convergence to v_π will be proven at the end of this section. We can choose to do *synchronous* backups.

Algorithm 3.1. Synchronous Backup

```
01.  for each iteration k+1:
02.        for all s ∈ S:
03.              update v_{k+1}(s) from v_k(s'), where s' is a successor state of s
```

We will discuss *acynchronous* backups later.

In synchronous backup, we update v_{k+1} as follows:

$$v_{k+1} = \mathcal{R}^\pi + \gamma \mathcal{P}^\pi v_k. \tag{3.2}$$

3. Policy Iteration

Now that we know how to evaluate a polic, we are going to update policy over and over to find a best policy.

Suppose that a policy π is given. How can we produce a new policy that is better than π?

Algorithm 3.2. A Single Policy Iteration

```
INPUT:   a policy π
01.  policy evaluation: evaluate π according to
```

$$v_\pi(s) = \mathbb{E}\left(R_{t+1} + \gamma R_{t+2} + \cdots | S_t = s\right). \tag{3.3}$$

```
02.  policy improvement: improve π by acting greedily with respect to v_π
```

In general, we need some (large) number of iterations until we converge to π_*, but the convergence is *guaranteed*.

This is what it means to act *greedily* with respect to v_π: given a deterministic policy $\pi(s)$, we define

$$\pi'(s) = \underset{a \in \mathcal{A}}{\operatorname{argmax}}\, q_\pi(s, a). \tag{3.4}$$

This improves the value from any state s over one step:

$$q_\pi\left(s, \pi'(s)\right) = \max_{a \in \mathcal{A}} q_\pi(s, a) \geq q_\pi\left(s, \pi(s)\right) = v_\pi(s). \tag{3.5}$$

Consequently, an inductive argument shows that

$$\begin{aligned} v_\pi(s) &\leq q_\pi\left(s, \pi'(s)\right) \\ &= \mathbb{E}_{\pi'}\left(R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s\right) \\ &\leq \mathbb{E}_{\pi'}\left(R_{t+1} + \gamma q_\pi\left(S_{t+1}, \pi'(S_{t+1})\right) | S_t = s\right) \\ &\leq \mathbb{E}_{\pi'}\left(R_{t+1} + \gamma R_{t+2} + \gamma^2 q_\pi\left(S_{t+2}, \pi'(S_{t+2})\right) | S_t = s\right) \\ &\leq \cdots \\ &\leq \mathbb{E}_{\pi'}\left(R_{t+1} + \gamma R_{t+2} + \cdots | S_t = s\right) = v_{\pi'}(s). \end{aligned}$$

In short:

$$v_\pi \leq v_{\pi'}. \tag{3.6}$$

When improvements stop, then [3.5] holds with an equality,

$$q_\pi\left(s, \pi'(s)\right) = \max_{a \in \mathcal{A}} q_\pi(s,a) = q_\pi(s, \pi(s)) = v_\pi(s), \tag{3.7}$$

which is precisely the Bellman optimality equation (see [2.9])

$$v_\pi(s) = \max_{a \in \mathcal{A}} q_\pi(s,a). \tag{3.8}$$

Thus it follows that $v_\pi = v_*$.

Often times even with a crude approximation of v_π, we can still obtain a very close approximation of π. This means much of the iterations onwards could become wasteful. Instead, we can impose a stopping condition. For instance, we can use ε-convergence: stop when the update in value function is less than π. More naively, we can fix some $k \in \mathbb{N}$ and stop after k iterations. An extreme case of this is when we have $k = 1$ – we update policy *every iteration*. We will see shortly that this is equivalent to *value iteration*, which is the topic of the next subsection.

4. Value Iteration

Any optimal policy can be subdivided into two components:

(a) an optimal first action A_*; and

(b) a following optimal policy from successor state S'.

Theorem 3.2. Principle of Optimality for Policies

Any policy $\pi(a|s)$ achieves the optimal value from state s,

$$v_\pi(s) = v_*(s),$$

if and only if for every state s' reachable from s, π achieves the optimal value on s', $v_\pi(s') = v_*(s)$.

If we know the solution to subproblems $v_*(s')$, then solution $v_*(s)$ can be found by one-step lookahead:

$$v_*(s) \leftarrow \max_{a \in \mathcal{A}} \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*\left(s'\right). \tag{3.9}$$

The idea of value iteration is to apply these updates iteratively: start with final rewards and work backwords. This still works with loopy, stochastic MDPs.

Unlike policy iteration, there is no explicit policy. Instead, we use synchronous backups (Algorithm 3.1),

Algorithm 3.3. Synchronous Backup

```
01.  for each iteration k+1:
02.        for all s ∈ S:
03.              update v_{k+1}(s) from v_k(s'), where s' is a successor state of s
```

to update value function:

$$v_1 \to v_2 \to \cdots \to v_*.$$

Convergence to v_* is guaranteed, and intermediate value functions $v_1, \ldots$ may not correspond to any policy. In vector form, the update ([3.9]) is given by

$$v_{k+1} = \max_{a \in A} \mathcal{R}^a + \gamma \mathcal{P}^a v_k. \tag{3.10}$$

5. Summary

problem	Bellman equation	algorithm
prediction	Bellman expectation equation	iterative policy evaluation
control	Bellman expectation equation and greedy policy improvement	policy iteration
control	Bellman optimality equation	value iteration

Algorithms are based on state-value function v_π or v_* with complexity $O\left(mn^2\right)$ per iteration for m actions and n states. The same idea also applies to action-value function q_π or q_*, but doing so has higher complexity $O\left(m^2n^2\right)$ per iteration.

6. Extensions to Dynamic Programming

DP methods described so far used *synchronous* backups (i.e. all states are backed up in parallel). *Asynchronous* DP backs up states individually, in any order: for each selected state, apply the appropriate backup. This can significantly reduce computation and is guaranteed to converge if all states continue to be selected.

We have three simple ideas for asynchronous DP:

(a) in-place DP;

(b) prioritized sweeping; and

(c) real-time DP.

In-place DP

Synchronous value iteration stores two copies, v_k, v_{k+1} of value function in any given time:

$$v_{k+1}(s) \leftarrow \max_{a\in\mathcal{A}} \mathcal{R}_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a v_k\left(s'\right). \tag{3.11}$$

In-place value iteration only stores one copy of value function:

$$v(s) \leftarrow \max_{a\in\mathcal{A}} \mathcal{R}_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a v\left(s'\right). \tag{3.12}$$

In case $v(s')$ is not updated, addition of the term $\gamma \mathcal{P}_{ss'}^a v(s')$ is identical to what is being done in [3.11]. Otherwise, if $v(s')$ is updated, then instead of using *old* value $v_k(s')$, we use the *updated* value $v(s')$.

Now the important part is to order the states so that the updates according to [3.12] can be done in an efficient manner.

Prioritized sweeping extends the idea of in-place DP.

Prioritized Sweeping

In in-place DP, we have seen that it is crucial to correctly set up the order of states. Prioritized sweeping achieves this by maintaining a priority queue of states, sorted by the decreasing order of Bellman error:

$$\left|\left(\max_{a\in\mathcal{A}} \mathcal{R}_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a v\left(s'\right)\right) - v(s)\right|. \tag{3.13}$$

Roughly speaking, we have:

Algorithm 3.4. Prioritized Sweeping

```
INPUT:  states (s_i)_{i∈I}
01. maintain a priority queue Q of (s_i)_{i∈I} according to [3.13]
02. while not done:
03.       backup the state with the largest remaining error
04.       update Bellman error of affected states
05.       update Q
```

For instance, when we are doing synchronous update (using 0 as the initial value for every state), it takes some time for the goal to propagate and change $v(s)$ to a nonzero value. To avoid thousands of *updates* on $v(s)$ from 0 to 0, we only choose the states where updating $v(s)$ is meaningful.

Real-time DP

The idea is to update only states that are relevant to agent (i.e. select only the states which the agent visits). Instead of updating every state naively, run the agent in the world, collect some samples, and update around those samples. That is, after each step (S_t, A_t, R_{t+1}), backup the state S_t according to

$$v(S_t) \leftarrow \max_{a \in \mathcal{A}} \left(\mathcal{R}^a_{S_t} + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{S_t s'} v(s') \right). \tag{3.14}$$

DP uses *full-width* backups. That is, for each (synchronous or asynchronous) backup, every successor state and action is considered using knowledge of the MDP transitions and reward function.

DP is effective for medium-sized problems with millions of states. But for large problems, DP suffers *Bellman's curse of dimensionality*: number of states $|\mathcal{S}|$ grows exponentially with number of state variables and even one backup becomes too expensive.

To resolve this issue, we use *sampling*. Instead of considering the whole branching factor, we simply sample few trajectories. This will be the topic we will consider shortly. We are going to use sample rewards and sampel transitions (S, A, R, S') instead of reward fucntion $\mathcal{R}$ and transition dynamics $\mathcal{P}$. This has few advantages:

(a) no advance knowledge of MDP is required; and *model-free*

(b) breaks the curse of dimensionality through sampling: cost of backup is constant, independent of $|\mathcal{S}|$.