Reinforcement Learning

Kristian Wichmann

June 28, 2018

1 Basics of reinforcement learning

Reinforcement learning deals with an *agent* making *decisions* over time in dealing with some *environment*, to maximize some cumulative, scalar *reward*.

An example would be playing a video game. Here, the player is the agent. The control inputs are the decisions. The game and its internal logic is the environment. And the score is the reward.

Sometimes, a reward will be greater if it is postponed - it is not always advantagegous to reap any immediate reward. In other words, reinforcement learning algorithms will not benefit from being greedy.

1.1 The reward hypothesis

The basis of all reinforcement learning is the *reward hypothesis*, which states:

> All goals can be described by the maximization of some cumulative, expected reward.

1.2 Observation, action, and reward

At at given timestep t, a reinforcement learning agent gets some input; an *observation* O_t about the environment. It then takes an *action* A_t. And finally it gets a scalar *reward* R_t.

1.3 History and state

At any given time step t, the agent has a *history* H_t, which simply consist of all the observations, actions, and rewards that have happened so far:

$$H_t = O_1, A_1, R_1, O_2, A_2, R_2, \cdots, O_t, A_t, R_t, \tag{1.1}$$

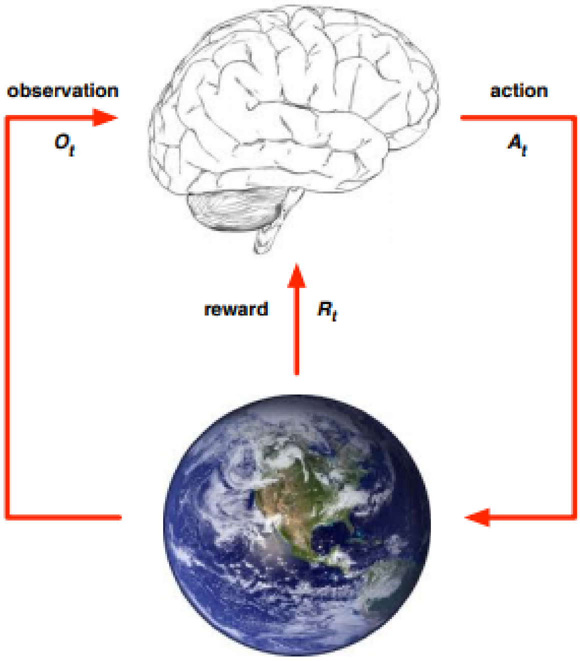


Figure 1: The interplay between agent and environment.

A *state* is a way for to parse this history into a more meaningful form. Formally, the state representation is simply a function of the history:

$$S_t = f(H_t) \tag{1.2}$$

It is very important to note, that this is generally distinct from the *environment state* H_t^e. The environment state contains complete information about the environment, including data and mechanisms that may be hidden to the agent. Hence H_t^e can depend on other things than just the history. Such information is *private* to the environment.

The *agent state* S_t^a on the other hand is the internal representation the agent uses to decide on which actions to take. Once again, it can be any function of history:

$$S_t^a = f(H_t) \tag{1.3}$$

1.4 Markov states

A state S_{t+1} is called a *Markov state* or an *information state* if it only depends on the state of the previous time step. Expressed probabilistically:

$$\mathbb{P}[S_{t+1}|S_1, S_2, \cdots S_t] = \mathbb{P}[S_{t+1}|S_t] \tag{1.4}$$

In other words, we don't need the entire history to decide on an action: Knowing the present is enough. Or put another way:

> The future is independent of the past, given the present.

Or:

> The state is a *sufficient statistic* of the future.

The environment state is always Markov, as by definition it contains all information about what can happen next. Similarly, the state consisting of the entire history is trivially Markov as well.

1.5 Full observability

This is the case where, in fact, we can observe everything about the environment and its inner workings. So that:

$$O_t = S_t^a = S_t^e \tag{1.5}$$

Sometimes this is reasonable. Sometimes not. But it will be a useful theoretical situation. This is known as a *Markov decision process* or MDP for short.

When this condition is not fulfilled, we speak about *partial observability* or a *partially observable evironment*. Here the agent only indirectly observes the invironment. This situation is known as a *partially observable Markov decision process* or POMDP for short.

1.6 State example: Bayesian beliefs

This state representation can be seen as a current best bet at what the actual environment state is. In other words, it is represented by Bayesian probabilities, which may then be updated over time using Bayes' rule:

$$S_t^a = (\mathbb{P}[S_t^e = s_1], \mathbb{P}[S_t^e = s_2], \cdots, \mathbb{P}[S_t^e = s_n]) \tag{1.6}$$

1.7 State example: Recurrent neural net

Here, the state S_t^a is a linear combination of the observation O_t and the state of the last time step S_{t-1}^a, followed by a non-linear *activation function* σ:

$$S_t^a = \sigma(W_O O_t + W_S S_{t-1}^a) \tag{1.7}$$

Here, the W's are weight matrices with sizes corresponding to the dimenstionality of observations and state vectors. Typical choices for σ are sigmoid, tanh, or rectified linear unit.

2 Components of an agent

A reinforcement learning agent may contain one or more of the following components:

- *Policy*: The agent's behaviour function. Shows how the agent gets from its state to deciding on an action.
- *Value function*: A measure of how desirable it is to be in a given state, or perform a given action.
- *Model*: The agent's representation of the environment.

We'll examine each of these in greater detail below:

2.1 Policy

A policy π is a mapping from state to action. For a *deterministic* policy, this is an ordinary function:

$$\pi(s) = a \tag{2.1}$$

So the state s is always mapped to the action a. We should ideal choose π so that the reward is maximized.

But a policy can also be *stochastic*, i.e. probabilistic. In this case π takes the form of conditional probabilities:

$$\pi(a|s) = \mathbb{P}[A = a|S = s] \tag{2.2}$$

2.2 Value function

A value function V_π is a prediction of the future reward for state s under a given policy π:

$$V_\pi(s) = \mathbb{E}_\pi[R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \cdots |S_t = s] \tag{2.3}$$

Here γ is a *discounting factor*, between 0 and 1. 0 means that we only care about the immediate reward, while values approaching 1 means that we care progressively more about long run rewards.

2.3 Model

In this context, a model tries to predict the evolution of the environment. These can take two different forms:

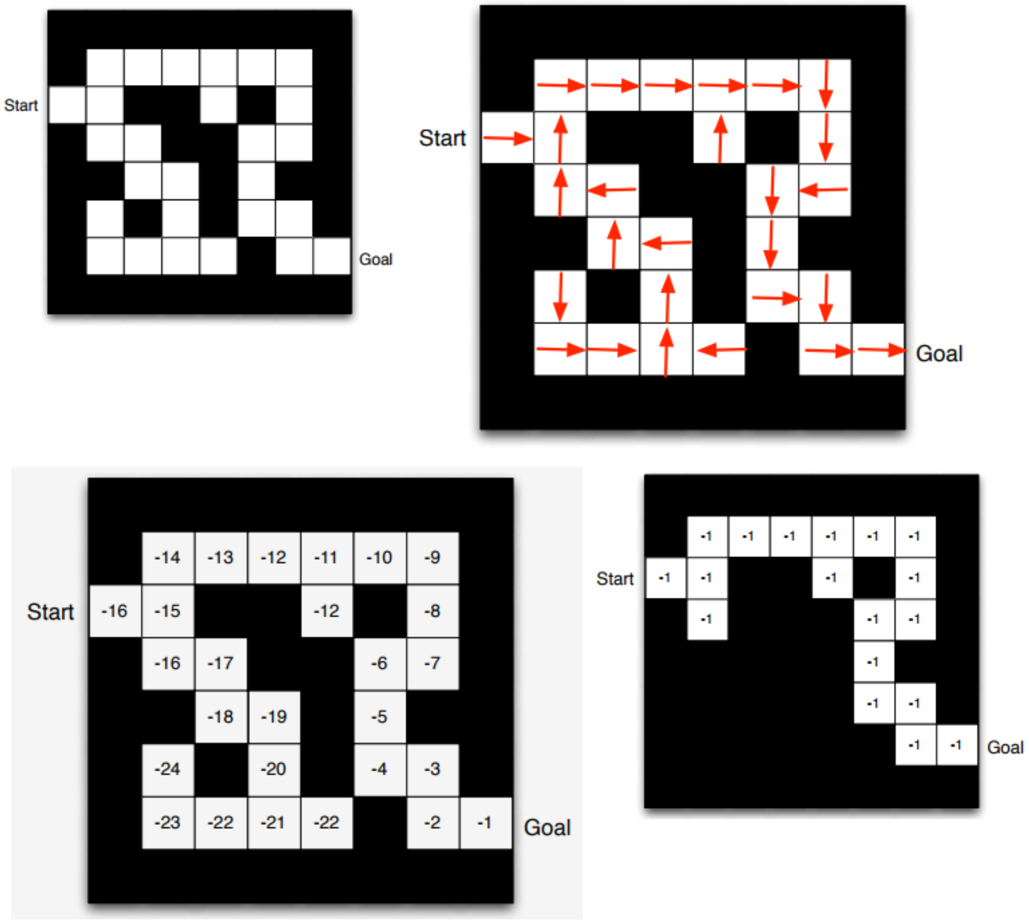


Figure 2: Upper left: Maze example. Upper right: Policy for maze. Lower left: Value function for maze. Lower right: Reward prediction for maze.

- *Transition prediction* $\mathcal{P}$ models the change of state of the environment:

$$\mathcal{P}^a_{ss'} = \mathbb{E}[S_{t+1} = s' | S_t = s, A_t = a] \tag{2.4}$$

- *Reward prediction* $\mathcal{R}$ models the change in (immediate) reward:

$$\mathcal{R}^a_s = \mathbb{E}[R_{t+1} | S_t = s, A_t = a] \tag{2.5}$$

2.4 Example: Maze

As an example, consider an agent that has to find its way though the maze in the upper left of figure 2. The problem has the following qualities:

- Rewards: We want the agent to get through the maze as quickly as possible, so each time step has a reward of -1.
- Actions: The agent can move up, down, left, or right in each time step.
- State: The state is the agents current position square in the maze.

A good policy for solving the problem is shown in the upper right. Here the arrows dictates the action given the state (position).

The lower left shows the value function corresponding to the policy: The future reward is negative the number of steps needed to get out of the maze from the current state (position). An optimal action can be chosen greedily, by picking the possibility with the lowest value of the function.

The lower right shows a reward prediction model for the maze. This one technically depends on both state and action, but as we no it is -1 no matter what.

2.5 Categorization of agents

Based on the above, we can broadly categorize agents into several categories:

- *Value based agents* are based on a value function - with the policy being implicit. I.e. in each step we can greedily pick the action with the highest future reward, based directly on the value function.
- *Policy based agents* on the other hand has the policy as its key ingredient. Here, the policy mapping directly shows what action to take in each state.
- *Actor critic agents* takes both a poicy and a value function into account. Ideally the "best of both worlds".
- *Model free agents* may by policy and/or value function based, but has no model! I.e. it doesn't try to make predictions about the environment.
- *Model agents* may by policy and/or value function based, but includes modelling.

3 Reinforcement learning and planning

There's two fundamental types of problems in sequential decision making: *Reinforcement learning* and *planning.*

Reinforcement learning can be viewed as dumping the agent into an alien environment it initially knows nothing about. The agent then begins to interact with the environment, eventually improving its policy for doing so.

Planning is the situation where the agent starts with a perfect model of the environment. The agent then uses this model to make calculations without any external interaction, which then informs policy.

So in planning, we can in principle look many steps ahead, determining states and rewards (or expectation values thereof) and use this to pick optimal actions. In other words, a tree search.

3.1 Exploration versus exploitation

So reinforcement learning is very much a trial-and-error process. The agent should hopefully learn an effective policy from interacting with the environment. This requires *exploration.* Ideally without losing too much reward along the way.

Exploration finds more information about the environment.

Planning, on the other hand relies on *exploitation* of the complete knowledge of the environment.

Exploitation uses known information to maximize reward.

Usually, a combination of exploration and exploitation is necessary for achieving success. So there's a balance between the two, sometimes knows as *exploration-exploitation tradeoff.* This consideration is unique to reinforcement learning as opposed to machine learning as a whole.

3.1.1 Examples

Situation: You're going out to eat.

- Exploitation: Going to your favorite restaurant.
- Exploration: Trying a new restaurant.

Situation: Online banner advertisement.

- Exploitation: Show the most succesful advert so far.
- Exploration: Display a different advert.

Situation: Drilling for oil.

- Exploitation: Drill at the best known location.
- Exploration: Drill at a new location.

3.2 Prediction and control

Prediction is trying out a given policy in practice and evaluating the results.

Control on the other hand is optimizing the future, i.e. finding the best strategy.

Usually in reinforcement learning, we have to do prediction in order to control.

4 Markov reward processes

Recall that a Markov decision process (or MDP for short) is a case where everything about the environment is known - the agent state is the same as the environment state. This also means that it has the Markov property.

It turns out that almost all reinforcement learning problems can be restated as MDP's. For instance:

- Optimal control primarily deals with continuous MDP's.
- Partially observable problems can be conversted into MDP's.
- *Bandits* are MDP's with one state.

In this section, we will deal with a simpler case - known as Markov reward processes - and the tackle the general problem later.

4.1 Transition probability matrix

Consider for a moment an agent with a state having the Markov property, which simply takes the same (or no) action in every time step. Then given the initial state s there is a probability it will end up in state s' in the next time step - it can depend only on s and s' per the Markov property:

$$\mathcal{P}_{ss'} = \mathbb{P}[S_{t+1} = s' | S_t = s] \tag{4.1}$$

These propabilities form a matrix; the *transition probability matrix* $\mathcal{P}$:

$$\mathcal{P} = \begin{pmatrix} \mathcal{P}_{11} & \mathcal{P}_{12} & \cdots & \mathcal{P}_{1n} \\ \mathcal{P}_{21} & \mathcal{P}_{22} & \cdots & \mathcal{P}_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ \mathcal{P}_{n1} & \mathcal{P}_{n2} & \cdots & \mathcal{P}_{nn} \end{pmatrix} \tag{4.2}$$

This form assumes a finite number of states n, but in principle this can be infinite.

Because the entries are probabilites, each row of $\mathbb{P}$ must sum to 1:

$$\forall i \in \{1, 2, \dots, n\} : \sum_{j=1}^{n} \mathcal{P}_{ij} = 1 \tag{4.3}$$

Such a system - represented by the tuple $\langle S, \mathcal{P} \rangle$ - is known as a *Markov process* or *Markov chain.*

4.2 Markov reward processes

A *Markov reward process* (or MRP) is a Markov process with a finite set of states and a *reward function* $\mathcal{R}$ as well as a discount factor $\gamma \in [0, 1]$. The reward function is defined as the expectation value of the next reward given the current state:

$$\mathcal{R} : s \mapsto \mathcal{R}_s = \mathbb{E}[R_{t+1} | S_t = s] \tag{4.4}$$

So a Markov reward process is the tuple $\langle S, \mathcal{P}, \mathcal{R}, \gamma \rangle$

4.3 Return and value function

The *return* of a given outcome of a Markov reward process[1] is:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{n=1}^{\infty} \gamma^{n-1} R_{t+n} \tag{4.5}$$

In other words, the return is the discounted future reward. This is the quantity we wish to maximize.

The value function, as we've already discussed above, is the expectation value of the return:

$$v(s) = \mathbb{E}[G_t | S_t = s] \tag{4.6}$$

[1] Or of any system with rewards and a discounting factor γ, really.

4.4 The Bellman equation for MRP's

We can now use equations 4.6, 4.5, and 4.4 to get a recursive formula for the value function:

$$
\begin{aligned}
v(s) =&\mathbb{E}[G_t|S_t = s] = && (4.7)\\
&\mathbb{E}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots |S_t = s] = && (4.8)\\
&\mathbb{E}[R_{t+1} + \gamma (R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + \cdots)|S = s] = && (4.9)\\
&\mathbb{E}[R_{t+1} + \gamma G_{t+1}|S_t = s] = && (4.10)\\
&\mathbb{E}[R_{t+1}|S_t = s] + \gamma \mathbb{E}[G_{t+1}|S_t = s] = && (4.11)\\
&\mathcal{R}_s + \gamma \sum_{s'=1}^{n} \mathbb{P}[G_{t+1}|S_{t+1} = s']\mathbb{P}[S_{t+1} = s'] = && (4.12)\\
&\mathcal{R}_s + \gamma \sum_{s'=1}^{n} \mathcal{P}_{ss'} v(s') && (4.13)
\end{aligned}
$$

Here, $\mathcal{R}_s$ is the reward for leaving state s - which might seem somewhat backwards. Summing it up:

$$v(s) = \mathcal{R}_s + \gamma \sum_{s'=1}^{n} \mathcal{P}_{ss'} v(s') \tag{4.14}$$

Treating the value and reward functions as vectors v and $\mathcal{R}$, this can be rewritten in matrix form:

$$v = \mathcal{R} + \gamma \mathcal{P} v \tag{4.15}$$

This equation can be solved using the usual methods:

$$v = \mathcal{R} + \gamma \mathcal{P} v \Leftrightarrow (I_n - \gamma \mathcal{P})v = \mathcal{R} \Leftrightarrow v = (I_n - \gamma \mathcal{P})^{-1} \mathcal{R} \tag{4.16}$$

This direct solution however, is only tractable for small n, as the run time for matrix inversion is $O(n^3)$. Instead, there is a number of other, faster possibilities, including:

- Dynamic programming.
- Monte-Carlo evaluation.
- Temporal difference learning.

4.4.1 Intuition behind the Bellman equation for MRP's

What is the intuition behind equation 4.14? Imagine the agent is in state s. We wish to calculate the expectation of the future discounted reward. First we leave s which gives an immediate reward of $\mathcal{R}_s$. To get the future reward we have to make a weighted average over all the possible states we can go to from s', each having a future reward of $v(s')$. Finally this is discounted by multiplying with γ.

5 Markov decision processes

Now for the full MDP treatment. This case is very similar to the MRP case, but now there is several different actions to choose from, collectively written as $\mathcal{A}$. To each action $a \in \mathcal{A}$, corresponds a transition probability matrix $\mathcal{P}^a$. So formally, a Markov decision process is a tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$, where:

- $\mathcal{S}$ is a finite set of states.
- $\mathcal{A}$ is a finite set of actions.
- $\mathcal{P}^a$ is a probability transition matrix corresponding to the action $a \in \mathcal{A}$.
- $\mathcal{R}^a_s$ is the expected reward for taking action $a \in \mathcal{A}$ when in state $s \in \mathcal{S}$.
- γ is the discounting factor.

5.1 Policy

Recall that a policy π is a way to choose an action $a \in \mathcal{A}$ given the state $s \in \mathcal{A}$:

$$\pi(a|s) = \mathcal{P}[A_t = a | S_t = s] \tag{5.1}$$

Theorem 5.1. *Given a Markov decision process $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$ and a policy π for it, then:*

- *The state sequence $S_1, S_2, \dots$ is a Markov process with probability transition matrix:*

$$\mathcal{P}^\pi_{ss'} = \sum_{a \in \mathcal{A}} \pi(a|s) \mathcal{P}^a_{ss'} \tag{5.2}$$

- *The combined state/reward sequence $S_1, R_1, S_2, R_2, \dots$ is a Markov reward process with reward function:*

$$\mathcal{R}^\pi_s = \sum_{a \in \mathcal{A}} \pi(a|s) \mathcal{R}^a_s \tag{5.3}$$

Proof. This follows from the law of total probability. □

This means that any MDP with a policy can always be reduced to a MRP, should we so wish.

5.2 State- and action-value function

The *state-value function* $v_\pi(s)$ of a MDP is the expected return starting from state s and then following policy π:

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] \tag{5.4}$$

The *action-value function* $q_\pi(s, a)$ is the expected return starting in state s, taking action a and then following policy π, i.e.:

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \tag{5.5}$$

5.3 Bellman's expectation equations

It turns out that state- and action-value functions obey recursive Bellman equations as well. First the state-value function, where we explicitly write out the sum over the possibilities of the first action of the π policy:

$$\begin{aligned} v_\pi(s) =& \mathbb{E}_\pi[G_t|S_t = s] = && (5.6)\\ & \sum_{a\in\mathcal{A}} \mathbb{P}[S_t = s, A_t = a]\mathbb{E}_\pi[G_t|S_t = s, A_t = a] = && (5.7)\\ & \sum_{a\in\mathcal{A}} \pi(a|s)q_\pi(s, a) && (5.8) \end{aligned}$$

So the state-value function involved the action-value function. Let's examine the latter:

$$\begin{aligned} q_\pi(s, a) =& \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = && (5.9)\\ & \mathbb{E}_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots |S_t = s, A_t = a] = && (5.10)\\ & \mathbb{E}_\pi[R_{t+1}|S_t = s, A_t = a]+ && (5.11)\\ & \quad \gamma\mathbb{E}_\pi[R_{t+2} + \gamma R_{t+3} + \cdots |S_t = s, A_t = a] = && (5.12)\\ & \mathcal{R}_s^a + \gamma\mathbb{E}_\pi[G_{t+1}|S_t = s, A_t = a] && (5.13) \end{aligned}$$

Now, write out the sum over that state at $t + 1$:

$$\begin{aligned} q_\pi(s, a) =& \mathcal{R}_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathbb{P}[S_{t+1} = s', S_t = s, A_t = a]\mathbb{E}_\pi[G_{t+1}|S_{t+1} = s'] = \\ & && (5.14)\\ & \mathcal{R}_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s') && (5.15) \end{aligned}$$

This means that we have a set of coupled equations:

$$v_\pi(s) = \sum_{a\in\mathcal{A}} \pi(a|s) q_\pi(s,a), \quad q_\pi(s,a) = \mathcal{R}_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s') \tag{5.16}$$

Inserting the latter into the former we get an equation for state-value function only:

$$v_\pi(s) = \sum_{a\in\mathcal{A}} \pi(a|s) \left[\mathcal{R}_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s') \right] \tag{5.17}$$

Similarly, we can get an equation for action-value function by inserting the former into the latter:

$$q_\pi(s,a) = \mathcal{R}_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a \left[\sum_{a'\in\mathcal{A}} \pi(a'|s') q_\pi(s',a') \right] \tag{5.18}$$

Do note the renaming of indices, though.

5.3.1 Intuition behind the Bellman expectation equations

What is the intuition behind the two equations 5.16?

Consider first the state-value function $v_\pi(s)$. Here, imagine the agent starting at state s. From there is can take a number of actions $a \in \mathcal{A}$, so we have to do a weighted average over the expected future rewards for each of these, the weights being $\pi(a|s)$. And the expected future rewards for being in state s and taking action a is exactly $q_\pi(s,a)$.

Consider then the action-value function $q_\pi(s,a)$. Here, the agent is in state s and takes action s. First, it gets the immediate reward $\mathcal{R}_s^a$ and then it goes to another state s', so we have to sum over these possibilities. The exptected future reward for being in state s' is $v_\pi(s')$. Finally, we must remember to discount the future reward by multiplying by γ.

5.4 Deterministic agents

A *deterministic* agent is one that always chooses a specific action $a(s)$ when in state s. Expressed in Kronecker delta form:

$$\pi(a|s) = \delta_{a,a(s)} \tag{5.19}$$

This means that the expectation equation for the state-value function simplifies:

$$v_\pi(s) = q_\pi(s, a(s)) \tag{5.20}$$

The intuition should be clear: The expected future reward in state s is simply the expected future reward for being in state s and taking action $a(s)$, since this is the only allowed action according to the policy.

We can now combine equation 5.20 with the action-value part of equation 5.16 to get:

$$v_\pi(s) = \mathcal{R}_s^{a(s)} + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^{a(s)} v_\pi(s') \tag{5.21}$$

5.5 Optimal value functions

The *optimal state-value function* $v_*(s)$ is the maximum[2] when considering all possible policies:

$$v_*(s) = \max_\pi v_\pi(s) \tag{5.22}$$

Similarly, the *optimal action-value function* $q_*(s, a)$ is:

$$q_*(s, a) = \max_\pi q_\pi(s, a) \tag{5.23}$$

5.6 Bellman's optimality equations

Let's imagine the agent is in the state s. It has a number of available strategies to choose from. Since it is optimal, it will have to choose the one that yields the greatest future outcome. This means that

$$v_*(s) = \max_a q_*(s, a) \tag{5.24}$$

I.e. the best future reward for being in state s must be achieved by taking the optimal action a. This is known as *Bellman's optimality equation.* We can use this along with the right hand side of equation 5.16 to get:

$$v_*(s) = \max_a \left[\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s') \right] \tag{5.25}$$

Or we can write the same equation, but this time for $q_*(s, a)$:

$$q_*(s, a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \left[\max_{a'} q_*(s', a') \right] \tag{5.26}$$

These non-linear equations are known as *Bellman's optimality equations.* In general these have no closed-form solution. Instead, typically iterative methods are used for solving them. Such strategies include:

[2] Or supremum, to be sure is exists.

- Value iteration.
- Policy iteration.
- Q-learning.
- Sarsa.

These will be examined later.

5.7 Optimal policies

We can define a partial ordering on policies as follows:

$$\pi \geq \pi' \Leftrightarrow \forall s \in \mathcal{S} : v_\pi(s) \geq v_{\pi'}(s) \tag{5.27}$$

Theorem 5.2. *For a Markov decision process, there exists an optimal policy π_*, i.e. on that is better than or as good as an arbitrary policy:*

$$\forall \pi : \pi_* \geq \pi \tag{5.28}$$

All such optimal policies achieves the optimal state-value and action-value function:

$$v_{\pi_*} = v_*, \quad q_{\pi_*} = q_* \tag{5.29}$$

Proof. We can construct such a policy by:

$$\pi_*(a|s) = \begin{cases} 1 & \text{if } a = \underset{a \in \mathcal{A}}{\text{argmax}}\ q_*(s,a) \\ 0 & \text{otherwise} \end{cases} \tag{5.30}$$

Since we must have $v_*(s) = \max\limits_a q_*(s|a)$ it follows that this policy achieves both optimal state-value and action-value function. □

This shows that not only does an optimal policy exist, but a deterministic, optimal policy exists!

6 Dynamic programming

Dynamic programming is a method for solving complex problems by breaking them down into subproblems, solving these and putting them back together to solve the overall problem.

In order for dynamic programming to be applicable, the problem must have two basic properties: *optimal substructure* and *overlapping subproblems.*

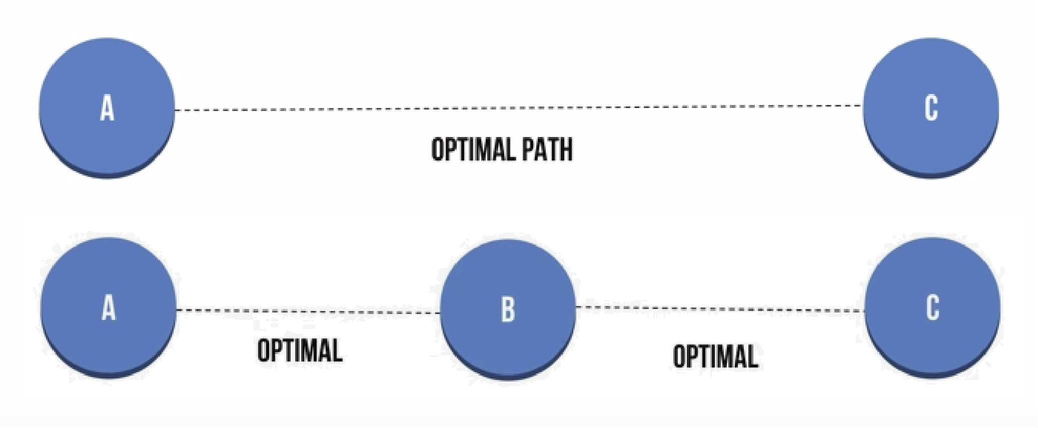


Figure 3: The principle of optimality shown for the case of finding optimal paths between points. The principle states, that we can find the optimal path from A to C (above) if we know the optimal paths from A to B and B to C (below).

6.1 Optimal substructure

This is also known as *the principle of optimality*. Assume we have an optimal solution for the case (A, C). Then that optimal solution should be obtainable from having the optimal solutions for the cases (A, B) and (B, C). Looking at figure 3 where the principle is illustrated for the problem of finding optimal paths.

Since the principle can be applied iteratively, eventually the problem can be reduced to a set of "smallest", subsequent subproblems. Because of this, mathematical induction is often used to prove a given problem has optimal substructure.

6.2 Overlapping subproblems

This means that each of the subproblems occur "many times" and that we therefore can cache the solutions from these so we don't have to redo the same problem over and over.

6.3 Example: Fibonacci numbers

As is well known, the Fibonacci numbers are defined recursively:

$$F_0 = 0, \quad F_1 = 1, \quad F_{n+1} = F_n + F_{n-1} \tag{6.1}$$

By this very definition, it is clear that if we know the all $F_i, i \leq n$ we can always find F_{n+1}, which means that the principle of optimality holds.

If we did this using only recursion, the fibonacci function would call itself a large number of times. We can prevent this by caching all F_n that is calculated. Thus the overlapping subproblem criterion is also satisfied.

7 The contraction mapping theorem

Before we continue exploring dynamc programming in the context of reinforcement learning, let's take a mathematical side trek which will be convenient for showing the convergence of certain algorithms.

Definition 7.1. *A contraction mapping on the metric space (X, d) is a Lipschitz function $T : X \to X$ with Lipschitz constant $q \in [0, 1[$, i.e.:*

$$\forall x, y \in X : \; d(T(X), T(Y)) \leq q \cdot d(x, y) \tag{7.1}$$

Theorem 7.1. *(The contraction mapping theorem) Let T be a contraction mapping on the non-empty, complete metric space (X, d). Then T has a unique fixed-point x^*:*

$$\exists! \; x^* \in X : \; T(x^*) = x^* \tag{7.2}$$

This fixed-point can be found by iteration of T from any starting point $x_0 \in X$:

$$x_n = T(x_{n-1}), \quad (x_n) \underset{n\to\infty}{\to} x^* \tag{7.3}$$

Proof. Using the triangle inequality for any $x, y \in X$ we have:

$$\begin{aligned} d(x, y) &\leq d(x, T(x) + d(T(x), T(y)) + d(T(y), y) & (7.4)\\ &\leq d(x, T(x) + q \cdot d(x, y) + d(T(y), y) & (7.5) \end{aligned}$$

Here we've used that T is a contraction. Isolate $d(x, y)$ to get:

$$d(x, y)(1 - q) \leq d(x, T(x) + d(T(y), y) \Leftrightarrow d(x, y) \leq \frac{d(x, T(x) + d(T(y), y)}{1 - q} \tag{7.6}$$

This is known as the *fundamental contraction inequality.* We can use it to prove that T has at most one fixed-point: Assume both x and y are fixed points. Then equation 7.6 says $d(x, y) \leq 0$, leaving a distance of zero as the only option, which means that $x = y$.

Now, consider the mapping $T^n : X \to X$ defined as repeated application of T n times. This mapping is also a contraction, with Lipschitz constant

q^n. We now wish to show that $(x_n) = (T^n(x_0))$ is Cauchy. We now apply the fundamental contraction inequality to x_m and x_n:

$$\begin{aligned}
d(x_m, x_n) &\leq \frac{d(x_m, T(x_m) + d(T(x_n), x_n)}{1-q} \tag{7.7}\\
&= \frac{d(T^m(x_0), T(T^m(x_0)) + d(T(T^n(x_0)), T^n(x_0))}{1-q} \tag{7.8}\\
&= \frac{d(T^m(x_0), T^m(T(x_0)) + d(T^n(T(x_0)), T^n(x_0))}{1-q} \tag{7.9}\\
&\leq \frac{q^m \cdot d(x_0), T(x_0)) + q^n \cdot d(T(x_0), x_0)}{1-q} \tag{7.10}\\
&= \frac{q^m + q^n}{1-q} d(x_0, T(x_0)) \tag{7.11}
\end{aligned}$$

Since $q < 1$, this can be made arbitrarily small by making m and n large enough. Hence (x_n) is Cauchy and therefore convergent, since X is complete. Let's call the limit x^*.

To show that x^* is a fixed-point consider the recursive definition of the series:

$$x_{n+1} = T(x_n) \tag{7.12}$$

Now apply the limit $n \to \infty$ on both sides of the equation:

$$\lim_{n\to\infty} x_{n+1} = \lim_{n\to\infty} T(x_n) \tag{7.13}$$

Since T is Lipschitz, it is also continuous, and so we may interchange limit and application of T:

$$\lim_{n\to\infty} x_{n+1} = T\left(\lim_{n\to\infty} x_n\right) \tag{7.14}$$

But we know that both these limits are equal to x^*, so:

$$x^* = T(x^*) \tag{7.15}$$

□

The theorem is also known as *Banach's fixed-point theorem.* By taking the limit $m \to \infty$ in the inequality in the proof, we get an estimate of the convergence speed towards x^*:

$$d(x^*, x_n) \leq \frac{q^n}{1-q} d(x_0, T(x_0)) \tag{7.16}$$

So in each step of the iteration process, we get closer to the fixed-point at a factor of q or lower.

8 Dynamic programming for MDP's

It turns out that we can use dynamical programming for finding value functions for MDP's. Here, the recursive substructure is implied by Bellman's optimality equations. And the value function can be cached as calculations are done. Hence, both criteria are satisfied.

This assumes perfect knowledge of the MDP, and therefore it can only be used for planning. There's essentially two types of question we can ask here:

- Prediction: Given a MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$ and a policy π, what is the corresponding state-value function $v_\pi(s)$?
- Control: Given a MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$, what is the optimal state-value function $v_*(s)$?

We'll look at both below.

8.1 Iterative policy evaluation

Evaluation relies on prediction: Given a policy π we want to find the state-value function $v_\pi(s)$. One approach - indeed the one we will use in this section - is to obtain the value function iteratively through Bellman's expectation equation 5.17. Here, we start by a guess v_1 - or simply zeros - at the state-value function as a vector, and then insert it into the expectation equation to get v_2. Iterating this process, getting v_3, v_4, v_5 etc., v_n will eventually converge to v_π:

$$v_1 \rightarrow v_2 \rightarrow v_3 \rightarrow \cdots \underset{n\rightarrow\infty}{\rightarrow} v_\pi \tag{8.1}$$

We will prove this later.

The update is done using *synchronous backup*, which means that we store the values of v_i for all the states and use them to calculate the values for v_{i+1}. Only then are the "active" values used in the next step calculation changed.

8.1.1 Example: Starting with $v_1 = 0$

Assume out inital guess v_1 simply consists of all zeros. Then the second iteration is:

$$v_2(s) = \sum_{a\in\mathcal{A}} \pi(a|s) \left[\mathcal{R}_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a v_1(s') \right] = \tag{8.2}$$

$$\sum_{a\in\mathcal{A}} \pi(a|s) \mathcal{R}_s^a \tag{8.3}$$

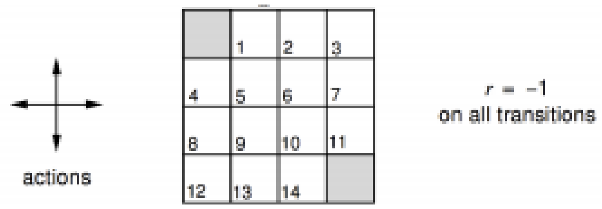


Figure 4: The gridworld environment. The grey areas make up the terminal state.

Similarly, the third iteration becomes:

$$v_3(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left[\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_2(s') \right] = \tag{8.4}$$

$$\sum_{a \in \mathcal{A}} \pi(a|s) \left[\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \sum_{a' \in \mathcal{A}} \pi(a'|s') \mathcal{R}_{s'}^{a'} \right] \tag{8.5}$$

And so on. Of course in practice this is done numerically.

8.1.2 Example: Gridworld

Consider the gridworld environment shown in figure 4. The agent's state is the square it occupies. The possible action are moving up, down, left, or right. The grey squares make up a terminal state. Trying to move across the outer border will make the agent stay where it is. In each step before it reaches the terminal area, the reward is -1.

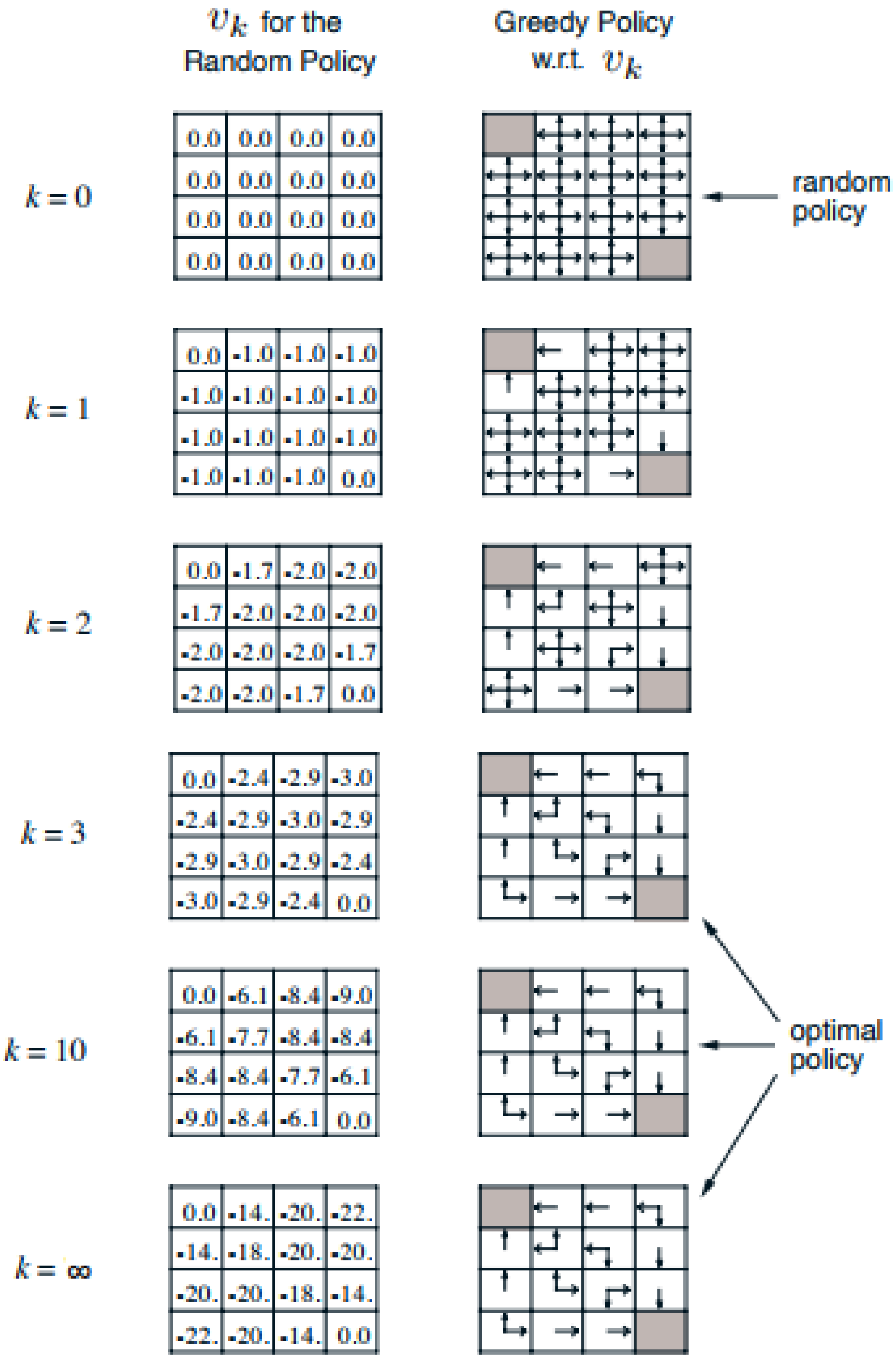
The policy π we wish to examine is uniform random choice between the four available actions in any non-terminal state:

$$\pi(\uparrow|\cdot) = \pi(\rightarrow|\cdot) = \pi(\downarrow|\cdot) = \pi(\leftarrow|\cdot) = \frac{1}{4} \tag{8.6}$$

The left part of figure 5 shows how v_k changes as the iterative policy evaluation process is performed. Eventually it converges; at $k = \infty$ the number at each square shows (save for the sign) the average number of steps a random walker takes before getting to the terminal state.

8.2 Convergence of iterative policy evaluation

So how are we sure that this iteration process actually converges? The answer has to do with the contraction mapping theorem.



$k = 0$

0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0
0.0	0.0	0.0	0.0

$k = 1$

0.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	-1.0
-1.0	-1.0	-1.0	0.0

$k = 2$

0.0	-1.7	-2.0	-2.0
-1.7	-2.0	-2.0	-2.0
-2.0	-2.0	-2.0	-1.7
-2.0	-2.0	-1.7	0.0

$k = 3$

0.0	-2.4	-2.9	-3.0
-2.4	-2.9	-3.0	-2.9
-2.9	-3.0	-2.9	-2.4
-3.0	-2.9	-2.4	0.0

$k = 10$

0.0	-6.1	-8.4	-9.0
-6.1	-7.7	-8.4	-8.4
-8.4	-8.4	-7.7	-6.1
-9.0	-8.4	-6.1	0.0

$k = \infty$

0.0	-14.	-20.	-22.
-14.	-18.	-20.	-20.
-20.	-20.	-18.	-14.
-22.	-20.	-14.	0.0

Figure 5: Left: Evolution of v_k starting at zero. Right: The greedy policy corresponding to v_k.

Here, we consider each configuration of state-value function v_n a point in euclidean space $X = \mathbb{R}^{|\mathcal{S}|}$ of dimension equal to the number of states. The distance we will use is the *Manhattan distance*, or d_∞ norm:

$$d(v, w) = \max_{s \in \mathcal{S}} |v(s) - w(s)| \tag{8.7}$$

In each iteration step, the operator $T : X \to X$ is applied, here written in matrix form:

$$T : v \mapsto \mathcal{R}^\pi + \gamma \mathcal{P} v \tag{8.8}$$

Given $v, w \in X$ we see:

$$\begin{aligned}
d(T(v), T(w)) &= \max_{s \in \mathcal{S}} |\mathcal{R}^\pi + \gamma \mathcal{P} v - (\mathcal{R}^\pi + \gamma \mathcal{P} w)| && (8.9)\\
&= \max_{s \in \mathcal{S}} |\gamma \mathcal{P} v - \gamma \mathcal{P} w| && (8.10)\\
&= \gamma \cdot \max_{s \in \mathcal{S}} |\mathcal{P}(v - w)| && (8.11)\\
&\leq \gamma \cdot d(v, w) && (8.12)
\end{aligned}$$

In the last step we used that none of the entries in $\mathcal{P}$ are larger than 1. So T is a contraction with Lipschitz constant q (assuming it is not exactly 1).

So by the contraction mapping theorem, the iterative process converges to a value of v which is a fixed-point for T, i.e.:

$$T(v) = v \Leftrightarrow \mathcal{R}^\pi + \gamma \mathcal{P} v = v \tag{8.13}$$

This is exactly the Bellman expectation equation, meaning the point of convergence is actually the state-value function for π.

8.3 Control: Policy iteration

Looking at the right hand side of figure 5, we see the *greedy policy* with respect to v_k. I.e. the policy that we choose the action for which the corresponding state is maximal (or if there's a tie, randomize even between those with maximal state-value).

These policies seem to be doing rather well. In fact, after only three iterations, it has arrived at the optimal policy, even if it takes way longer for v_k to converge to v_π! This should give us an idea for how to improve our policy π:

1. Start with a policy π.
2. Find v_π using iterative policy evaluation.

3. Construct the policy $\pi' = \text{greedy}(v_\pi)$.

The new policy will always be better (or at least as good) as π:

$$\pi' \geq \pi \tag{8.14}$$

We will show this in the next section. If the process is applied iteratively, it will always converge to an optial policy π_*. This is known as *policy iteration*. Again, this will be shown in the next section.

For the gridworld example, only one such iteration was needed to reach an optimal policy. This is rarely the case for more complicated problems.

8.4 Convergence of iterative policy improvement

So why does all of this work? Let's start by considering only deterministic policies. Since greedy policies are deterministic (or can be made deterministic while retaining efficiency), this can be done without loss of generality.

So let π be a deterministic policy, i.e.:

$$\pi(s) = a(s) \tag{8.15}$$

Now make a new policy π' by being greedy with respect to v_π. Recall, that according to Bellman's expectation equation:

$$v_\pi(s) = \sum_{a\in\mathcal{A}} \pi(a|s) q_\pi(s,a) = q_\pi(s, a(s)) \tag{8.16}$$

Here we have used that π is deterministic. So being greedy corresponds to maximizing $q_\pi(a,s)$. This means:

$$\pi'(s) = \underset{a\in\mathcal{A}}{\text{argmax}}\ q_\pi(s,a) \tag{8.17}$$

Why does this improve the policy? Let's see:

$$q_\pi(s, \pi'(s)) = \max_{a\in\mathcal{A}} q_\pi(s,a) \geq q_\pi(s, a(s)) = v_\pi(s) \tag{8.18}$$

In the last equal sign, we used equation 8.16 again.

It turns out, that when equation 8.18 is satisfied we can apply the following useful theorem, the *theorem of policy improvement*:

Theorem 8.1. *(Theorem of policy improvement) Given two deterministic policies, such that for all states $q_\pi(s, \pi'(s)) \geq v_\pi(s)$. Then $\pi' \geq \pi$.*

Proof. Use the inequality assumption and the definiton of q_π to get:

$$v_\pi(s) \leq q_\pi(s, \pi'(s)) = \mathbb{E}_{\pi'}[R_t + \gamma v_\pi(S_{t+1})|S_t = s] \tag{8.19}$$

Here we've used that the expectation value of a constant is simply the constant itself, meaning that $\gamma v_\pi(S_{t+1}) = \mathbb{E}_{\pi'}[\gamma v_\pi(S_{t+1})]$. But now we may use 8.18 again:

$$v_\pi(s) \leq \mathbb{E}_{\pi'}[R_t + \gamma q_\pi(S_{t+1}, \pi'(s))|S_t = s] \tag{8.20}$$

But now we can apply the same steps again to get:

$$v_\pi(s) \leq \mathbb{E}_{\pi'}[R_t + \gamma R_{t+1} + \gamma^2 v_\pi(S_{t+2})|S_t = s] \tag{8.21}$$

This can be iterated indefinitely, so in the end we get:

$$\begin{aligned} v_\pi(s) \leq & \mathbb{E}_{\pi'}[R_t + \gamma R_{t+1} + \gamma^2 R_{r+2} + \gamma^3 R_{r+3} + \cdots |S_t = s] = & (8.22) \\ & \mathbb{E}_{\pi'}[G_{t+1}|S_t = s] = v_{\pi'}(s) & (8.23) \end{aligned}$$

This shows that $\pi \leq \pi'$. □

Now assume that we perform policy iteration until there is no further improvement. I.e. until:

$$v_\pi = v_{\pi'}, \quad q_\pi = q_{\pi'} \tag{8.24}$$

In this case, we can rewrite equation 8.18, but this time with an equal sign:

$$q_\pi(s, \pi'(s)) = \max_{a \in \mathcal{A}} q_\pi(s, a) = q_\pi(s, a(s)) = v_\pi(s) \tag{8.25}$$

But this means that π satisfies the Bellman optimality equation:

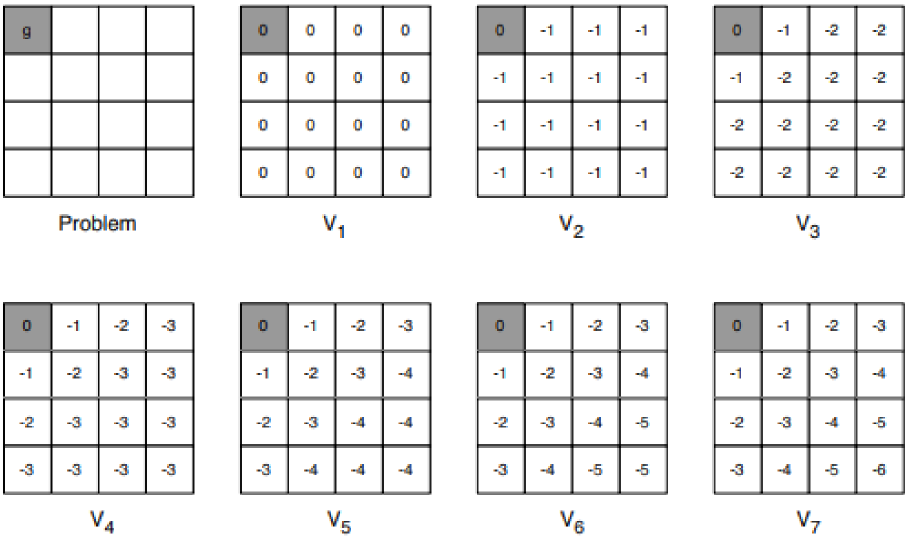
$$v_\pi(s) = \max_{a \in \mathcal{A}} q_\pi(s, a) \tag{8.26}$$

Hence π is optimal. This shows convergence of iterative policy improvement.

8.5 Modified policy iteration

As we noticed in the gridworld example, sometimes we don't really need the full convergence of v_* to get a good policy by acting greedy.

Modified policy iteration exploits this idea by only taking k steps in each policy evaluation loop.



g			

Problem

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

V_1

0	-1	-1	-1
-1	-1	-1	-1
-1	-1	-1	-1
-1	-1	-1	-1

V_2

0	-1	-2	-2
-1	-2	-2	-2
-2	-2	-2	-2
-2	-2	-2	-2

V_3

0	-1	-2	-3
-1	-2	-3	-3
-2	-3	-3	-3
-3	-3	-3	-3

V_4

0	-1	-2	-3
-1	-2	-3	-4
-2	-3	-4	-4
-3	-4	-4	-4

V_5

0	-1	-2	-3
-1	-2	-3	-4
-2	-3	-4	-5
-3	-4	-5	-5

V_6

0	-1	-2	-3
-1	-2	-3	-4
-2	-3	-4	-5
-3	-4	-5	-6

V_7

Figure 6: Left: Evolution of v_k starting at zero. Right: The greedy policy corresponding to v_k.

8.6 Value iteration

This method explicity exploits the principle of optimality: An optimal strategy for an agent in state s necesserily starts with a first optimal action a_*. If we know this, we can also choose the optimal action a'_* in any state s' the agent may end up in after taking action a_*. This is essentially what is expressed in Bellman's optimality equation:

$$v_*(s) = \max_a \left[\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s') \right] \tag{8.27}$$

Following the idea of iterative policy evaluation, we can start at some value of $v_*(s)$ and iterate this equation. This is known as *value iteration.*

Since we're essentially being greedy in every update step, this is equivalent to modified policy iteration with $k = 1$.

8.6.1 Example: More gridworld

As an example, consider the gridworld from earlier, except with only one terminal square. The possible actions at each non-terminal square is still the four directions. But this time, we search for a value function instead of evaluating a policy. Figure 6 shows the state-value function as value iteration is performed starting from a uniform value of zero. We see that each step propagates the reward through the state space until convergence after 7 steps.

9 Asynchronous dynamic programming

So far, all the iterative processes have been synchronous. I.e. we've update all the states of the value functions at the same time. However, there's advantages to updating the values *asynchronously*. The convergence will often become faster, and assuming all states are updated it is still guaranteed.

The rest of this section is a quick overview of three such strategies.

9.1 In-place dynamic programming

Here, one value at a time is updated as described above. The order in which the states are chosen is usually random, making sure all states are updated, and that no patterns affect the convergence process.

9.2 Prioritised sweeping

Not all states have the same impact on the convergence of the algorithms. As we saw in the gridword value iteration example in figure 6 change in value functions "ripple" out from certain states. The idea here, is the note how much the value changed in the previous update. Then sort them based on the absolute value of the change and do the updates in this order. A state that has changed a lot tends to have more effect on subsequent updates, so with this approach, we should get more change in values overall.

This can be implemented using a priority queue data sctructure.

9.3 Real-time dynamic programming

Here, we focus on updating the states that are most relevant for the agent. This is done by simulating an agent moving through the state space, and then only updating states it comes into contact with.

10 Model-free prediction

So far we have focused on planning. In other words, the situation where er know all the details of the environment. Obviously, this is not alwys the case.

In this section we will consider the predicion problem in the case of an MDP we don't know the details of. This is known as *model-free prediction*.

10.1 Monte Carlo learning

To use what is known as the *Monte Carlo (MC) learning* method, we must have a process which terminates at some point - the method only learns from what is known as *full episodes.*

So given a policy π, we can start an agent in state $S_1 \in \mathcal{S}$ and then let the process run, following π, getting rewards and so on, until the episode terminates at step k. The total episode can then be summed up as:

$$S_1, A_1, R_1, S_2, A_2, R_3, \ldots, S_k \tag{10.1}$$

At any step in the episode t, we can then calculate the discounted reward:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^{k-t-1} R_{k-1} \tag{10.2}$$

Recall that the state-value function is the expected value of this:

$$v_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s] \tag{10.3}$$

According to the law of large numbers, if we take the average of a large number of actualized values, we will converge to the expected value with probability 1. In other words, if we run this a lot of times $n \gg 1$, each time starting in state $S_1 = s$ and getting discounted reward G_1^i (i.e. calculated from step 1) in the i'th run, we have:

$$v_\pi(s) \approx \frac{1}{n} \sum_{i=1}^{n} G_1^i \tag{10.4}$$

Now, of course, we need not start at position s. If we end up at s at any time during the episode, we can n different approaches.

10.1.1 First-visit MC evaluation

Here, to estimate $v_\pi(s)$ from a set of n episodes (with random starting states), we count how many episodes s is reached $N(s)$. For each such episode, we calculate the discounted reward from the $firstvisit$ to s in the episode G^j_{first}. We may now estimate the state-value function as:

$$v_\pi(s) \approx \frac{1}{N(s)} \sum_{j=1}^{N(s)} G^j_{\text{first}} \tag{10.5}$$

10.1.2 Every-visit MC evaluation

But of course, because of the Markov property, there's not fundamental difference between the first time we visit state s in an episode, from any subsequent times. In other words, we might as well include all the times s is visited in any episode. That total number of visits plays the part of $N(s)$ here. The discounted reward is calculated from each of these $G^j_{t_j}$. The formula for the approximate state-value function is then:

$$v_\pi(s) \approx \frac{1}{N(s)} \sum_{j=1}^{N(s)} G^j_{t_j} \tag{10.6}$$

10.2 Incremental mean

In either of the two cases above, instead of waiting until we've found all the relevant visits and then averaging, we might as well have done a running calculation of the mean.

To see how this works, consider a sequence of observations $x_1, x_2, \ldots$. For the k'th step, we may calculate the mean for the first k values:

$$\mu_k = \frac{1}{k} \sum_{i=1}^{k} x_i \tag{10.7}$$

Our goal is to calculate μ_k just from μ_{k-1} and x_k. Rewrite equation 10.7 as follows:

$$\mu_k = \frac{1}{k} \left(\sum_{i=1}^{k-1} x_i + x_k \right) \tag{10.8}$$

$$= \frac{1}{k}((k-1)\mu_{k-1} + x_k) \tag{10.9}$$

$$= \mu_{k-1} + \frac{1}{k}(x_k - \mu_{k-1}) \tag{10.10}$$

One way to think of this, is that μ_{k-1} is the best estimate so far. $x_k - \mu_{k-1}$ is how much the new observation is off, and so we adjust μ_k to fit it.

10.2.1 Incremental mean for Monte Carlo method

Here, when we get to the k'th (first if first-visit) visit to state s, with a discounted reward G_k, the update can be written:

$$v_\pi(s) \approx V_k(s) = V_{k-1}(s) + \frac{1}{k}(G_k - V_{k-1}(s)) \tag{10.11}$$

Here, we use the notation of $V_k(s)$ for the guess after the $k'th$ observation.

10.2.2 Incremental mean for non-stationary processes

As we can see, we continually need to keep track of how many measurements we have done so far to use incremental means. However, if the process is *non-stationary*, i.e. if it eventually changes over time, we don't really want to consider episodes that are too far in the past. In this case, we may alter the incremental mean formula to:

$$v_\pi(s) \approx V_k(s) = V_{k-1}(s) + \alpha(G_k - V_{k-1}(s)) \tag{10.12}$$

Here we don't need to keep track of k. Instead, we essentially assume, that only the last $1/\alpha$ visits are important.

10.3 Temporal-difference learning

Temporal-difference (TD) learning is another model-free policy evaluation method. However, in contrast to MC, TD can learn from *incomplete episodes.* This has the advantage that the process need not always terminate. This is done by what is known as *bootstrapping.*

10.4 TD(0)

If we look at equation 10.12, we can interpret is as follows:

$$\text{new value} = \text{old value} + \alpha(\text{estimated reward} - \text{old value}) \tag{10.13}$$

Previously, we've estimated the future discounted reward by actually calculating it from the terminating episode. But we could also estimate it is:

$$\text{immediate reward} + \gamma \cdot \text{future reward} \tag{10.14}$$

For the future reward, our best guess is the currently estimated value. So, in other words, this gives us the following update rule for our estimate:

$$V(S_t) \mapsto V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)) \tag{10.15}$$

This method is known as *TD(0).* We will make the following two definitions:

- $R_{t+1} + \gamma V(S_{t+1})$ is known as the *TD target.*
- $\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$ is called the *TD error.*

Since the TD target is an estimate of the return - as opposed to MC where it is the actually observed return - the TD error will have an expectation value that differs from zero. The TD target is a *biased* estimator of the return.

10.5 Bias-variance tradeoff

Let's consider the properties of the MC vs. TD(0) learning estimates.

For MC learning we consider many time steps - whole episodes. Since this is an ordinary average, the estimate is unbiased. However, we will have a tendency to overfit to the actually observed episodes, exactly since we're considering long stretches of time. To see why: Consider an agent which goes through an episode and eventually "dies" (terminates). If this was a human, and we're trying to estimate the cause of death, it is possible that recent events (not looking both ways before crossing the road) was the cause of death, as opposed to things that happened long time ago (a food poisoning 3 years ago). Since everything is considered, the algorithm will often find patterns where there are none. In other words, it will overfit, i.e. have a high variance. But since everything is considered, given enough data, eventually it will find the true causes - it is unbiased.

On the other hand, for TD learning, we only consider the very recent actions/rewards in each step. In the analogy above, this means that we are more likely to identify immediate causes/rewards. Long term patterns are very hard to find, but on the other hand, it doesn't see patterns where there are none. So the algorithm has a high bias, but low variance.

Another view is that MC is based on a lot of sample points, which means that there's a lot of aggregated noise, but on the other hand, having many points will make sure we're close to the real value. For TD, the situation is reversed: Only one point is used, meaning that there's only noise from one point, but it is also hard to estimate true value.

Luckily, it turns out that there is a way to choose an algoritm that is a compromise between these two extremes, which allows us to tune the bias-variance tradeoff to suit our needs.

10.6 Convergence properties of MC and TD

But before that, let's look a little deeper into the differences between MC and TD learning.

It turns out, that both MC and TD will converge to v_π as the number of episodes tends to infinity. The question is how fast this happens. Figure 7 shows the RMS error for both methods (and some different values of α) for a random walk model. As we can see, TD tends to converge faster than MC. For both methods, higher values of α are more "jittery", as high learning rates has a greater tendency to over/under-shoot in each step.

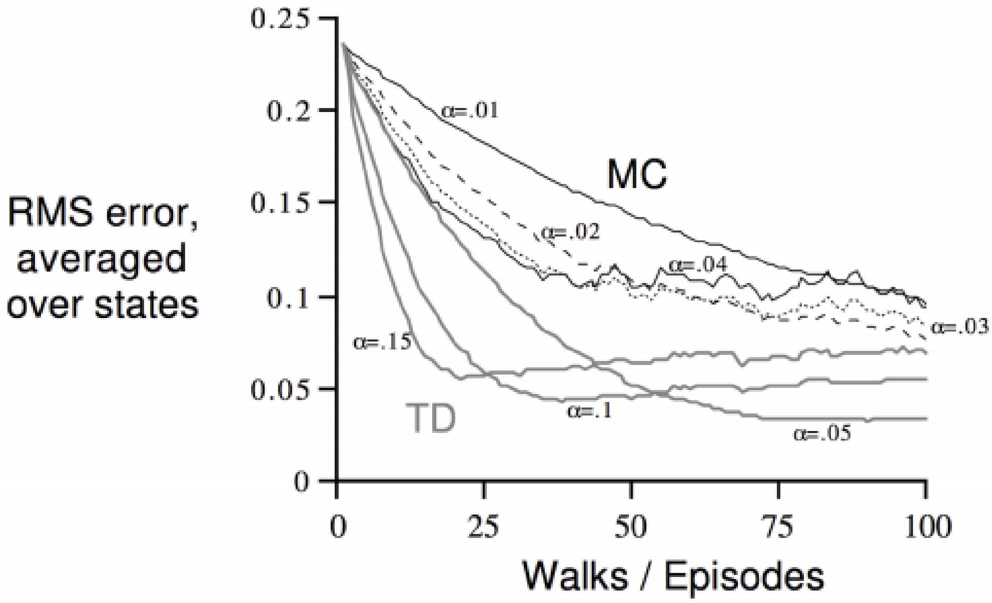


Figure 7: Comparing convergence of MC and TD for a random walk.

10.7 Batch MC and TD

Above, we assumed that we had access to an arbitrarily large amount of sample data. But what happens if we have just a finite set of K episodes - a *batch* - to train the data on?

In this case, we can instead keep using the limited set of data, randomly picking from these until convergence. How can we characterize the resulting value functions in each case?

For MC, we get $v_\pi(s)$ equal to the average of G_t for all data points where $S_t = s$. Alternatively, we may characterize this as the v_π which minimizes the sum of squared errors:

$$\sum_{k=1}^{K} \sum_{t=1}^{T_k} \left(G_t^k - v_\pi(s_t^k)\right)^2 \tag{10.16}$$

For TD, when convergence is reached, the TD errors must be zero:

$$\mathbb{E}\left[v_\pi\left(s_t\right)\right] = \mathbb{E}\left[R_{t+1} + \gamma v_\pi\left(s_{t+1}\right)\right] \tag{10.17}$$

Here, s_t and s_{t+1} can be any consecutive states in any of the episodes, and the expectation value is taken over the batch. This looks similar to a Markov

Decision Process. In this case we search for a model which satisfies:

$$v_\pi(s) = \sum_{a\in\mathcal{A}} \pi(a|s) \left[\mathcal{R}_s^a + \gamma \sum_{s\in\mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s')\right] \tag{10.18}$$

We can now consider this a probablistic model with v_π as a random variable, and $\mathcal{R}$ and $\mathcal{P}$ as parameters. Then the maximum likelihood model is the one which averages the parameters over the relevant occurences in the batch:

$$\mathcal{R}_s^a = \frac{1}{N(s,a)} \sum r_s^a, \quad \mathcal{P}_{ss'}^a = \frac{1}{N(s,a)} \sum \delta_{s's_n} = \frac{N(s,a,s')}{N(s,a)} \tag{10.19}$$

Here $N(s,a)$ is the number of times state s is followed by action a in the batch, and the sums are over these as well. s_n is the state after s (the next state), so the second sum counts the number of times s, a, s' occurs. When averaging over the batch, this assures that equation 10.18 is satisfied.

10.7.1 Example

Consider a system with two states, A and B. There is no choice between actions. The batch consists of the following 8 episodes:

- $A, 0, B, 0$.
- $B, 1$ (6 times).
- $B, 0$.

The question now is: What is $v(A)$ and $v(B)$? ($\gamma = 1$).

For Monte Carlo, this is an average over the return for each state. So:

$$v(A) = \frac{0}{1} = 0, \quad v(B) = \frac{0+1+1+1+1+1+1+0}{8} = \frac{3}{4} \tag{10.20}$$

For TD(0), we build the corresponding MDP. First $\mathcal{R}$:

$$\mathcal{R}_A = \frac{0}{1} = 0, \quad \mathcal{R}_B = \frac{0+1+1+1+1+1+1+0}{8} = \frac{3}{4} \tag{10.21}$$

Then $\mathcal{P}$:

$$\mathcal{P}_{AB} = \frac{1}{1} = 1, \quad \mathcal{P}_{BA} = 0 \tag{10.22}$$

Also, from B there's a probability of 1 of ending the episode. The corresponding MDP is visualized in figure 8. From this we infer that just like before $v(B) = 3/4$, but $v(A) = 3/4$ as well!

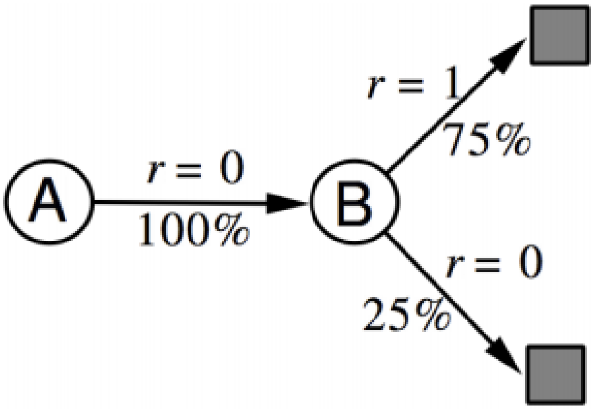


Figure 8: The example MDP build on batch.

10.8 n-step temporal-difference learning

So far, the temporal difference has only been one step into the future. But there's nothing preventing us from us looking more than one step ahead in our estimates, changing the TD target as appropriate. We can look n steps ahead:

$$G^{(1)}(t) =R_{t+1} + \gamma V(S_{t+2}) \tag{10.23}$$
$$G^{(2)}(t) =R_{t+1} + \gamma R_{t+2} + \gamma^2 V(S_{t+2}) \tag{10.24}$$
$$\vdots \quad \vdots \tag{10.25}$$
$$G^{(\infty)}(t) =R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots \tag{10.26}$$

Notice that the limiting case is the same as Monte Carlo learning, as all subsequent steps are taken into account.

So the update rule for n-step temporal-difference learning becomes:

$$V(S_t) \mapsto V(S_t) - \alpha(G^{(n)} - V(S_t)) \tag{10.27}$$

This is very similar to the gradient descent algorithm for training neural networks. Here, α plays the part of the learning rate, while the parenthesis can be thought of as an error (similar to a derivative).

10.9 TD(λ)

So $n = 1$ and $n = \infty$ are two extremes corresponding to TD(0) and MC respectively. Meaning that low n has high bias, low variance. And high n

has low bias, high variance. It turns out that there is a better way of tuning the tradeoff, though.

Instead of using simply one n, a sum over all values of n is used. This is done by using a weight $\lambda \in [0, 1]$. The target is then set to:

$$G_t^\lambda = (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_t^{(n)} \tag{10.28}$$

This known as the λ-*return*. The weights add up to 1, because of the geometric series:

$$\sum_{n=0}^{\infty} \lambda^n = \frac{1}{1 - \lambda} \tag{10.29}$$

Hence $1 - \lambda$ acts as a normalization constant. If the episode terminates, we assign the remaining weight to the highest possible value of n.

We see that when $\lambda = 0$ we do indeed get TD(0). When $\lambda = 1$, and the episode terminates, all but the last weight is zero, and we get MC.

10.10 Overview of RL algorithms

Let's pause for a moment and consider the strategies of the algorithms we've seen so far.

10.10.1 Backup

Figures 9 and 10 show different approaches to the *backup* strategy used so far. The first axis in figure 9 shows the *depth* of the backup, while the second axis shows the *width* of the backup.

Consider first the brute force approach of an exhaustive search. Here all of the data are used - not merely samples - which means that the backup is *full.* In every step of iteration, the entire path back to the starting point is used, which means that the backup is *deep.*

Dynamic programming still uses full backups - all data are used. But here, in each step, the backup is *shallow*, as we only look one step forward in the data.

Monto Carlo learning is based on a sample, and hence uses *sample* backup. But it uses the data for all of the episode, so the backup is also deep.

TD(0) learning uses sample backup as well. But since we only look one step ahead, the backup is shallow.

Finally, TD(λ) learning uses sample backups too. But it allows us to move along the first axis, from shallow to deep backup as λ varies.

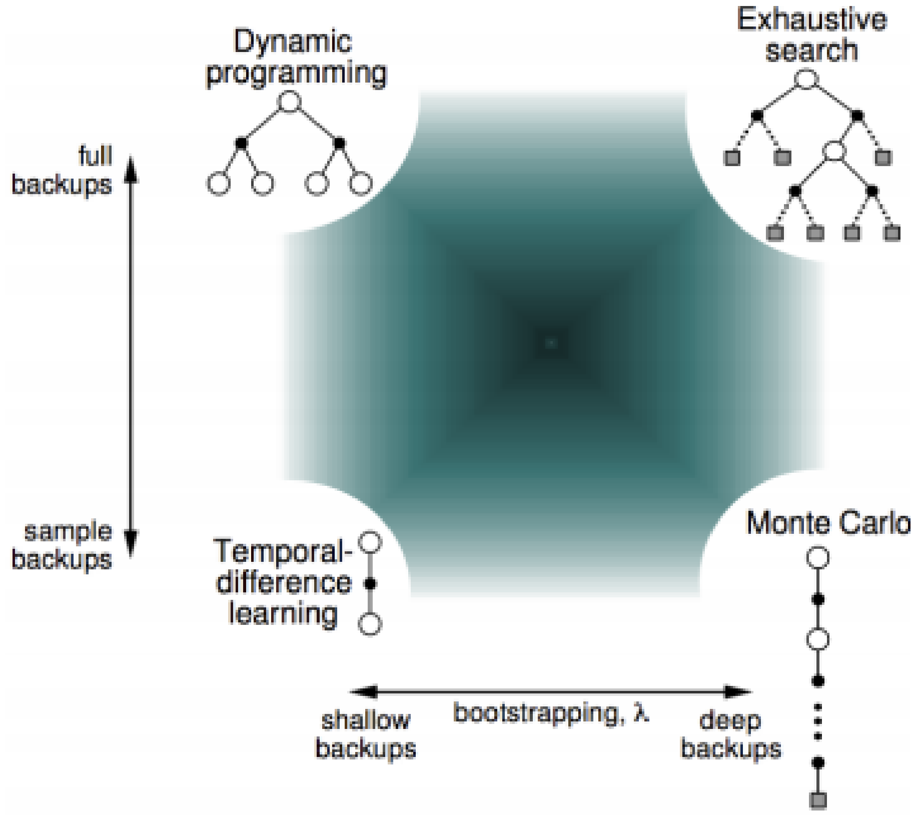


Figure 9: Backup space for algorithms.

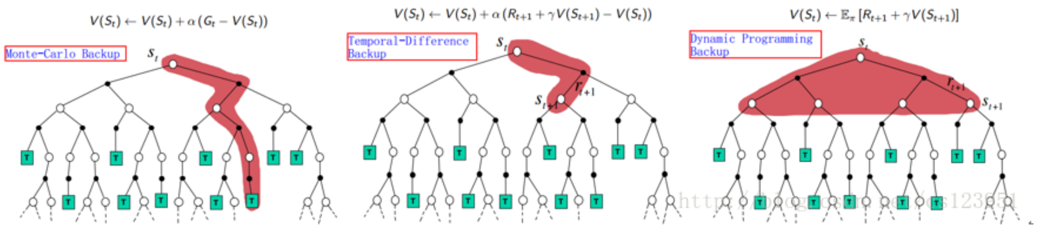

$V(S_t) \leftarrow V(S_t) + \alpha(G_t - V(S_t))$
Monte-Carlo Backup
S_t
T
$V(S_t) \leftarrow V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t))$
Temporal-Difference Backup
S_t
r_{t+1}
S_{t+1}
$V(S_t) \leftarrow \mathbb{E}_\pi[R_{t+1} + \gamma V(S_{t+1})]$
Dynamic Programming Backup
S_t
r_{t+1}
S_{t+1}

Figure 10: Illustration of backups for MC, TD(0), and DP respectively. An exhaustive search would cover all of the tree.

10.10.2 Bootstrapping and sampling

In statistics, a *bootstrap* is the process of resampling data. In RL, an algorithm is said to use bootstrapping, if the update involves an estimation based on previous prediction.

Dynamic programming bootstraps, as seen from the Bellman equation. Temporal difference learning bootstraps as well. Monte Carlo learning does not, as it is instead based on the entire episode sample.

An algorithm that does not involve an expectation value, i.e. an estimate including all possible future outcomes is said to *sample*. Monte Carlo and TD samples, while dynamic programming does not.

10.10.3 Online vs. offline learning

An algorithm can be operate *online* or *offline*. In this context it depends on when the updates to the value functions are done. Online learning means that values are changed while the episode still runs - it is done after every single action. Offline learning, on the other hand is done after the entire episode is finished.

10.11 Backward view: Eligibility traces

So far we have been taking a *forward view*, looking n steps forward into the process and so on. However, in practice, we want to do updates based on what went before our current time step. To do so, we use the tool of *eligibility traces* for TD(λ):

At time step zero, we set the *eligibilty* E of all states equal to zero: $E_0(s) = 0$. These are then updated in each time step according to the following rules:

- All eligibilities are reduced as follows: $E_t(s) = \lambda\gamma E_{t-1}(s)$
- If $S_t = s$, then in addition to the discounting above, 1 is added to eligibility. So all in all: $E_t(s) = \lambda\gamma E_{t-1}(s) + 1$

After this, $V(s)$ is updated for all s, so that the change is proprotional to $E_t(s)$ and the SD error δ_t:

$$\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \tag{10.30}$$

Update rule (for all s) is:

$$V(s) \mapsto V(s) - \alpha\delta_t E_t(s) \tag{10.31}$$

10.12 Equivalence between views

It turns out that the forward and backward views are equivalent, at least for offline learning, i.e. when the updates are only done once the episode is over.

Theorem 10.1. *The sum of offline updates for state s is the same for forward and backward views in TD(λ):*

$$\sum_{t=1}^{T-1} \alpha \left(G_t^{(\lambda)} - V(S_t) \right) \cdot 1_{S_t = s} = \sum_{t=1}^{T-1} \alpha \delta_t E_t(s) \tag{10.32}$$

Here $1_{S_t=s}$ is equal to 1 when $S_t = s$, and 0 otherwise.

Proof. Let's start by the special cases of the extreme values of λ.

When $\lambda = 0$, the eligibility traces are all reset each step. Which means that for a time step t where $S_t = s$, $E_t(s) = 1$, and otherwise all elegibility traces are zero. This is exactly equivalent to (offline) TD(0) learning.

On the other hand, if $\lambda = 1$, the eligibility traces decay only according to γ. This means that the eligibilty trace for a state $E_t(s)$ keeps track of how many times the state s has been visited up to and including time t. First, let's consider the case where s is visited once, at time step k. This means that the eligibility function is:

$$E_t(s) = \begin{cases} 0 & \text{for } t < k \\ \gamma^{t-k} & \text{for } t \geq k \end{cases} \tag{10.33}$$

This means that the right hand sum is:

$$\sum_{t=1}^{T-1} \alpha \delta_t E_t(s) = \alpha \sum_{t=k}^{T-1} \gamma^{t-k} \delta_t \tag{10.34}$$

Using the definition in equation 10.30 this is:

$$\alpha \sum_{t=k}^{T-1} \gamma^{t-k} \left[R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \right] \tag{10.35}$$

Writing out each term in the sum:

$$\begin{aligned}
&R_{k+1} + \gamma V(S_{k+1}) - V(S_k) && (10.36)\\
&+\gamma R_{k+2} + \gamma^2 V(S_{k+2}) - \gamma V(S_{k+1}) && (10.37)\\
&+\gamma^2 R_{k+3} + \gamma^3 V(S_{k+3}) - \gamma^2 V(S_{k+2}) && (10.38)\\
&+\cdots && (10.39)\\
&+\gamma^{T-1-k} R_T + \gamma^{T-k} V(S_T) - \gamma^{T-1-k} V(S_{T-1}) && (10.40)
\end{aligned}$$

A lot of terms now cancel leaving

$$R_{k+1}+\gamma R_{k+2}+\gamma^2 R_{k+3}+\cdots+\gamma^{T-1-k}R_T-V(S_k) = G_k^{(1)}-V(S_k) = \delta_k \quad (10.41)$$

Here we've also used that $V(S_T) = 0$ since the state is terminal. So in the case where s is visited once, we get (first-visit) Monte Carlo learning result. If s is revisited, there's wil be a contribution for each visit. But since elegibility traces are additive with respect to several visits, the equality remains true.

The general case is very similar to the $\lambda = 1$ calculation. Again, assume that s is visited once, at time k. Now, the eligibility trace is:

$$E_t(s) = \begin{cases} 0 & \text{for } t < k \\ (\lambda\gamma)^{t-k} & \text{for } t \geq k \end{cases} \quad (10.42)$$

The right hand sum now becomes:

$$\alpha \sum_{t=k}^{\infty} (\lambda\gamma)^{t-k} \left[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\right] \quad (10.43)$$

Since the episode is no longer required to terminate, we can sum to infinity here. Write out the sum, and move terms to make time steps line up:

$$\begin{aligned} -V(S_k) + & R_{k+1} + \gamma V(S_{k+1}) - \lambda\gamma V(S_{k+1}) && (10.44) \\ & +\lambda\gamma \left[R_{k+2} + \gamma V(S_{k+2}) - \lambda\gamma V(S_{k+2})\right] && (10.45) \\ & +(\lambda\gamma)^2 \left[R_{k+3} + \gamma V(S_{k+3}) - \lambda\gamma V(S_{k+3})\right] \cdots && (10.46) \end{aligned}$$

Pull out factors of $1 - \lambda$:

$$\begin{aligned} -V(S_k) + & R_{k+1} + \gamma(1-\lambda)V(S_{k+1}) && (10.47) \\ & +\lambda\gamma \left[R_{k+2} + \gamma(1-\lambda)V(S_{k+2})\right] && (10.48) \\ & +(\lambda\gamma)^2 \left[R_{k+3} + \gamma(1-\lambda)V(S_{k+3})\right] + \cdots && (10.49) \end{aligned}$$

Now remember that:

$$\sum_{n=0}^{\infty} \lambda^n = \frac{1}{1-\lambda} \quad (10.50)$$

Use this to rewrite:

$$R_t = (1-\lambda)\frac{R_t}{1-\lambda} = (1-\lambda)\sum_{n=0}^{\infty} R_t \lambda^n \quad (10.51)$$

So we can also pull out factors of $1 - \lambda$ from the rewards, so that we get (here is sum notation):

$$(1-\lambda)\left[\sum_{m=0}^{\infty}(\lambda\gamma)^m \left(\sum_{n=0}^{\infty} R_{k+m}\lambda^n + \gamma V(S_{k+m+1})\right)\right] - V(S_k) \quad (10.52)$$

Offline updates	$\lambda = 0$	$\lambda \in (0,1)$	$\lambda = 1$
Backward view	TD(0)	TD(λ)	TD(1)
	$\parallel$	$\parallel$	$\parallel$
Forward view	TD(0)	Forward TD(λ)	MC
Online updates	$\lambda = 0$	$\lambda \in (0,1)$	$\lambda = 1$
Backward view	TD(0)	TD(λ)	TD(1)
	$\parallel$	$\nparallel$	$\nparallel$
Forward view	TD(0)	Forward TD(λ)	MC
	$\parallel$	$\parallel$	$\parallel$
Exact Online	TD(0)	Exact Online TD(λ)	Exact Online TD(1)

Figure 11: Comparison of forward/backward views and offline/online updates for TD learning.

We now wish to group the contents of the parenthesis by powers of λ. If we want all terms of power p, then we must have $p = m + n$. Or $m = p - n$. Remembering that $m \geq 0$, this means:

$$(1-\lambda)\sum_{p=0}^{\infty}\lambda^p\left[\sum_{n\leq p}^{\infty}\gamma^n R_{k+p-n} + \gamma^{p+1}V(S_{k+p-n+1})\right] - V(S_k) \tag{10.53}$$

On closer inspection, this is:

$$G_k^\lambda - V(S_k) \tag{10.54}$$

For more than one visit to $s = S_k$, this generalizes as above. □

11 Model-free control

Last section dealt with evaluating a policy, but usually what we really want it control - finding an optimal policy. Again, we will take a model-free approach. This is useful, because even if the underlying model is a MDP, we may not know it. Or we may know it, but it's too big to handle without sampling.

11.1 General policy improvement theorem

So far, we have only shown that the policy improvement theorem applies to deterministic policies. In fact, it also applies when the policies are stochastic:

Theorem 11.1. *Given two stochastic policies, such that for all states:*

$$q_\pi(s, \pi'(s)) \equiv \sum_{a \in \mathcal{A}} \pi'(a|s) q_\pi(s, a) \geq v_\pi(s) \tag{11.1}$$

Then $\pi' \geq \pi$.

Proof. The starting point is:

$$v_\pi(s) \leq \sum_{a \in \mathcal{A}} \pi'(a|s) q_\pi(s, a) \tag{11.2}$$

The action-value function q_π can be written:

$$q_\pi(s, a) = \mathbb{E}_\pi[R_t + \gamma v_\pi(S_{t+1})|S_t = s, A_t = a] \tag{11.3}$$

Use $\pi(a|s)$ to expand the expectation value:

$$q_\pi(s, a) = \sum_{a' \in \mathcal{A}} \pi(a'|s) \left(\mathcal{R}_s^{a'} + \gamma \sum_{s'} \mathcal{P}_{ss'}^{a} v_\pi(s') \right) \tag{11.4}$$

Here $\mathcal{P}$ is associated with policy π. Reinstate the a sum:

$$\sum_{a \in \mathcal{A}} \pi'(a|s) \sum_{a' \in \mathcal{A}} \pi(a'|s) \left(\mathcal{R}_s^{a'} + \gamma \sum_{s'} \mathcal{P}_{ss'}^{a} v_\pi(s') \right) \tag{11.5}$$

This can be expressed as an expectation value over π' and π respectively:

$$\begin{aligned} &\mathbb{E}_{\pi'} \left[\sum_{a' \in \mathcal{A}} \pi(a'|s) \left(\mathcal{R}_s^{a'} + \gamma \mathbb{E}_\pi[v_\pi(S_{t+1})] \right) \middle| S_t = s \right] && (11.6) \\ =&\mathbb{E}_{\pi'} \left[\mathbb{E}_\pi \left[R_t + \gamma \mathbb{E}_\pi[v_\pi(S_{t+1})] \right] | S_t = s \right] && (11.7) \\ =&\mathbb{E}_{\pi'} \left[R_t + \gamma v_\pi(S_{t+1}) | S_t = s \right] && (11.8) \end{aligned}$$

But now we may use the inequality assumption once again:

$$\sum_{a'' \in \mathcal{A}} \pi'(a''|S_{t+1}) q_\pi(S_{t+1}|a'') \leq v_\pi(s') \tag{11.9}$$

Similarly to the original proof, we can now repeat this process above once more to get:

$$v_\pi(s) \leq \mathbb{E}_{\pi'} \left[R_t + \gamma R_{t+1} + \gamma^2 v_\pi(S_{t+2}) \middle| S_t = s \right] \tag{11.10}$$

This can be iterated indefinitely, so just as in the original proof we get:

$$\begin{aligned} v_\pi(s) \leq &\mathbb{E}_{\pi'}[R_t + \gamma R_{t+1} + \gamma^2 R_{r+2} + \gamma^3 R_{r+3} + \cdots | S_t = s] = && (11.11) \\ &\mathbb{E}_{\pi'}[G_{t+1}|S_t = s] = v_{\pi'}(s) && (11.12) \end{aligned}$$

□

This version of the theorem will come in handy in a short while.

11.2 On- and off-policy learning

We may learn about policy π by actually following the policy π (when doing control, we want to improve it along the way). This is known as *on-policy* learning. But we may also learn about π by following another policy μ - "looking over the shoulder" of it, so to speak.

At first we will focus on on-policy learning.

11.3 Generalized policy iteration

We want to mimic the strategy of policy iteration, generalizing it to the model-free case. Recall that policy iteration consists of two components:

- Policy evauation: We can use - say MC learning - to estimate v_π for the current policy π.
- Policy improvement: Improve the model by acting greedily with respect to v_π.

Each of these components are problematic in the model free case:

- Acting greedily with respect to v_π requires knowledge of the underlying MDP model.
- Also, when we're acting greedily, we may cut off parts of the state space which has a long-time reward payoff. Which means we can get stuck at a local minimum.

The solution to the first problem is to use q_π instead. In this case, for a state s, we can simply pick the action a with the hightest value of q_π.

The second problem can be avoided by acting ε-*greedily.*

11.4 ε-greedy exploration

A strategy which ensures that we will eventually explore every part of the state space is the ε-greedy improvement.

Before getting to these, let's look at a related definition: A discrete, finite probability distribution is called ε-*soft*, if the probability of any singleton event is greater than or equal to ε/m, where m is the number of possible outcomes.

Now for the ε-greedy strategy: When improving the policy π, the new policy π' has a probability ε of picking a random action (uniformly chosen). Otherwise it acts greedy as usual. Such a policy is called ε-greedy as well.

Similarly, a policy which has an ε-soft distribution for choosing between actions, is itself called ε-soft. I.e. when it satisfies:

$$\pi(a|s) \geq \frac{\varepsilon}{|\mathcal{A}|} \tag{11.13}$$

Specifically, an ε-greedy policy is ε-soft.

It turns out, that the ε-greedy strategy always produces a new policy π' at least as good as π, provided π is itself ε-soft:

Theorem 11.2. *If π is an ε-soft policy, then the new policy π' made by being ε-greedy with respect to q_π is at least as good as π: $v_{\pi'}(s) \geq v_\pi(s)$.*

Proof. We wish to use the general policy improvement theorem, so consider:

$$\begin{aligned} q_\pi(s, \pi'(s)) &= \sum_{a\in\mathcal{A}} \pi'(a|s) q_\pi(s,a) && (11.14)\\ &= \frac{\varepsilon}{m}\sum_{a\in\mathcal{A}} q_\pi(s|a) + (1-\varepsilon)\max_{a\in\mathcal{A}} q_\pi(s|a) && (11.15)\end{aligned}$$

Here we have assumed there m possible actions. But the maximum must be greater than any weighted sum of q_π's, assuming the weights sum to one. We will choose the weights:

$$w_a = \frac{\pi(a|s) - \varepsilon/m}{1-\varepsilon} \tag{11.16}$$

Because π is ε-soft, these are all non-negative. These weights sum to one:

$$\sum_{a\in\mathcal{A}} w_a = \sum_{a\in\mathcal{A}} \frac{\pi(a|s)}{1-\varepsilon} - \sum_{a\in\mathcal{A}} \frac{\varepsilon/m}{1-\varepsilon} = \frac{1}{1-\varepsilon} - \frac{\varepsilon}{1-\varepsilon} = \frac{1-\varepsilon}{1-\varepsilon} = 1 \tag{11.17}$$

So:

$$\begin{aligned} q_{\pi'}(s, \pi') &\geq \frac{\varepsilon}{m}\sum_{a\in\mathcal{A}} q_\pi(s|a) + (1-\varepsilon)\sum_{a\in\mathcal{A}} \frac{\pi(a|s) - \varepsilon/m}{1-\varepsilon} q_\pi(s|a) && (11.18)\\ &= \frac{\varepsilon}{m}\sum_{a\in\mathcal{A}} q_\pi(s|a) + \sum_{a\in\mathcal{A}} \pi(a|s) q_\pi(s|a) - \frac{\varepsilon}{m}\sum_{a\in\mathcal{A}} q_\pi(s|a) && (11.19)\\ &= \sum_{a\in\mathcal{A}} \pi(a|s) q_\pi(s|a) = v_\pi(s) && (11.20)\end{aligned}$$

By the general policy improvement theorem we have $\pi' \geq \pi$. □

11.5 Monte Carlo control

We can now use the strategy above using Monte Carlo learning. As always this requires episodes which terminates. The steps are as follows:

- Use Monte Carlo learning to estimate q_π for the current policy π, running episodes until convergence.
- Update policy ε-greedily and repeat.

However, as we saw in the last section, usually we need not wait until convergence in the first step before updating the policy. We may simply update after a set number of episodes. Taken to the extreme, we may update the policy after every episode.

11.6 Greedy in the Limit with Infinite Exploration

We want to make sure that we actually converge to the optimal policy π_* in the iteration process described above. The following definition is useful:

Definition 11.1. *An policy iteration strategy is called Greedy in the Limit with Infinite Exploration (GLIE) if the following two conditions are met:*

- *Every possible state/action is asymptotically explored an infinite number of times:*
$$\lim_{k\to\infty} N_k(s,a) = \infty \tag{11.21}$$
- *The policy converges to a greedy policy:*
$$\lim_{k\to\infty} \pi_k(a|s) = \delta_{a,a(s)} \tag{11.22}$$

  Here, $a(s)$ must be $\lim_{k\to\infty} \underset{a'\in\mathcal{A}}{\mathit{argmax}}\, q_{\pi_k}(s,a')$.

  This can be achieved by decreasing ε during the iteration.

11.7 GLIE Monte Carlo Control

Consider the following policy iteration strategy:

- Initialize $N(s,a) = 0$ for all state-action pairs.
- Repeat the following steps:
  - Sample an episode $\{S_1, A_1, R_2, S_2, A_2, \ldots, S_T\}$ according to π.

- Add 1 to $N(s,a)$ for every $S_n = s, A_n = a$ pair in the episode.
- Update $q(S_t, A_t)$ for every state-action pair in the episode:

$$q(S_t, A_t) \mapsto q(S_t, A_t) + \frac{1}{N(S_t, A_t)}(G_t - q(S_t, A_t)) \tag{11.23}$$

- Let $\varepsilon = 1/k$, where k is the episode number. Update π to acting ε-greedy with respect to $q(s,a)$.

11.8 TD(0) Control: Sarsa

The Monte Carlo method - as usual - has the disadvantage that it can only be updated offline, i.e. after one or more episode have terminated.

So instead we turn to SD-inspired strategies. The simplest one these is *sarsa*, which - somewhat unimaginatively - is short for: ”State, Action, Reward, State, Action”.

The idea is: Given a state S and an action A chosen according to the current policy π. We get a reward R and end up in state S'. We then decide on the next action A', again using the current policy π. Then update q according to:

$$q(S,A) \mapsto q(S,A) + \alpha(\underbrace{R + \gamma q(S', A')} - q(S,A)) \tag{11.24}$$

Here, the underbraced part is the bootstrapped estimate of $q(S,A)$, and the entire paranthesis is the SD error. Next, update π to be ε-greedy with respect to the new values of q.

So here, we update the policy after every single time step, looking one step ahead. This is analogous to TD(0) for policy evaluation.

Now, the big question is: Does following this strategy converge to an optimal policy? It turns out that is does, assuming the GLIE conditions are met. These turn out to be equivalent to the two following criteria: The first crititerion then comes down to:

$$\sum_{k=1}^{\infty} \varepsilon_k = \infty, \quad \sum_{k=1}^{\infty} \varepsilon_k^2 < \infty \tag{11.25}$$

For instance, $\varepsilon_k = 1/k$ results in a convergent GLIE strategy.

11.9 n-step sarsa

Just as was the case for evaluation, instead of just looking one step ahead, we could look n steps ahead and modify the SD estimate accordingly:

$$q_t^{(n)} = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{n-1} R_{t+n-1} + \gamma^n q(S_{t+n}, A_{t+n}) \tag{11.26}$$

The update rule then becomes:

$$q(S_t, A_t) \mapsto q(S_t, A_t) + \alpha(q_t^{(n)} - q(S_t, A_t)) \tag{11.27}$$

11.10 λ-sarsa (forward view)

Following evaluation once again, we wish to consider a sum of different n-step sarsas, to gather the best qualities of different lookaheads, so to speak. So here the SD estimate is:

$$q_t^\lambda = (1-\lambda)\sum_{n=1}^{\infty} \lambda^{n-1} q_t^{(n)} \tag{11.28}$$

And the update rule for q is:

$$q(S_t, A_t) \mapsto q(S_t, A_t) + \alpha(q_t^\lambda - q(S_t, A_t)) \tag{11.29}$$

This is the forward view: We look to future events to calculate q_t^λ.

11.11 λ-sarsa (backward view)

We can again use eligibility traces to formulate an equivalent view of λ-sarsa. This time looking back in time, so that it may be easily applied online.

At each episode, eligibility traces are reset: $E_0(s, a) = 0$. At each step thereafter, each are discounted:

$$E_t(s, a) = \gamma\lambda E_{t-1}(s, a) \tag{11.30}$$

Finally, one is added to the eligibility corresponding to the current state-action pair:

$$E(S_t, A_t) \mapsto E(S_t, A_t) + 1 \tag{11.31}$$

After each step, all $q(s, a)$ are updated according to SD-error and eligibility. SD-error is:

$$\delta_t = R_{t+1} + \gamma q(S_{t+1}, A_{t+1}) - q(S_t, A_t) \tag{11.32}$$

The update rule is:

$$q(s, a) \mapsto q(s, a) + \alpha\delta_t E_t(s, a) \tag{11.33}$$

12 Off-policy learning

Doing off-policy learning is like "looking over the shoulder" of some agent using the strategy μ, even though we want to optimize policy π.

This approach may also be used if we want to re-use previously played data - like older versions of the policy π earlier in the policy iteration.

We might even want to learn about several policies while following μ.

12.1 Change of distribution for expectation value evaluation

Imagine we want to calculate the expectation value of a random variable $f(X)$, where X is itself a random variable with distribution $P(X)$. But we want to express this in terms of another distribution $Q(X)$. We can make this change be the following rewrite:

$$\begin{aligned} \mathbb{E}_{X\sim P}[f(X)] &= \sum_X P(X)f(X) && (12.1)\\ &= \sum_X Q(X)\frac{P(X)}{Q(X)}f(X) && (12.2)\\ &= \mathbb{E}_{X\sim Q}\left[\frac{P(X)}{Q(X)}f(X)\right] && (12.3) \end{aligned}$$

The case where P and Q corresponds to differenct policies, like μ and π is known as *importance sampling*

12.2 Monte Carlo importance sampling

In this case, we want to use an entire episode of following μ to improve π. But since each time step is a different stochastic variable, we must change distribution for each step:

$$G_t^{\pi/\mu} = \frac{\pi(A_t|S_t)}{\mu(A_t|S_t)}\frac{\pi(A_{t-1}|S_{t-1})}{\mu(A_{t-1}|S_{t-1})}\cdots\frac{\pi(A_T|S_T)}{\mu(A_T|S_T)}G_t \qquad (12.4)$$

We can then update the policy for π according to:

$$V(S_t) \mapsto V(S_t) + \alpha(G_t^{\pi/\mu} - V(S_t)) \qquad (12.5)$$

One thing is that this will not work if π is non-zero when μ is zero, but in general the many factors means that the variance will be huge. In practice, this is no good!

12.3 TD(0) importance sampling

Instead we turn our attention to temporal-difference learning. For TD(0), we look one time step forward. For off-policy learning, this means we only have to use importance sampling for this single step to evaluate the SD target:

$$V(S_t) \mapsto V(S_t) + \alpha \left(\frac{\pi(A_t|S_t)}{\mu(A_t|S_t)} (R_{t+1} + \gamma V(S_{t+1})) - V(S_t) \right) \tag{12.6}$$

This means that there's much lower variance than for MC, even though it may still be problematic.

12.4 Q-learning

The above is off-policy learning for the state-value function V. We now turn to the subject of learning the action-value function $Q(s, a)$.

It turns out that we can avoid importance sampling in this case. The strategy is that given the sequence S_t, A_t, S_{t+1}, we use μ to choose the next action A_{t+1}. But we also choose an action A' based on π! Now update Q but use A' for bootstrapping:

$$Q(S_t, A_t) \mapsto Q(S_t, A_t) + \alpha(R_{t+1} + \gamma Q(S_{t+1}, A') - Q(S_t, A_t)) \tag{12.7}$$

Then, after each such step, we update both π and μ by the following strategy:

- π is updated to a greedy strategy with respect to $Q(s, a)$.
- μ is updated to a ε-greedy strategy with respect to $Q(s, a)$.

This means that the Q-learning target is:

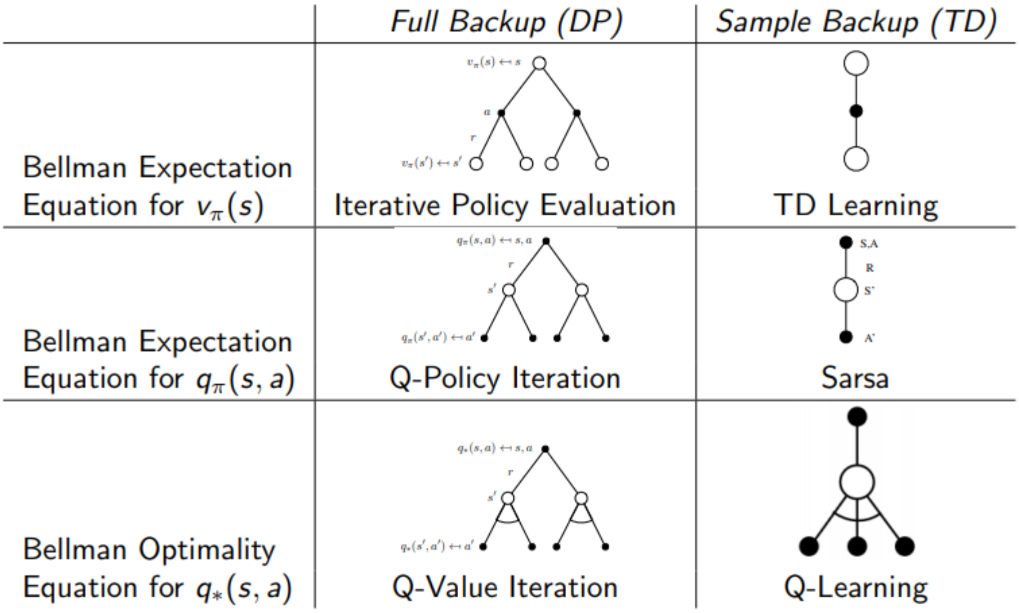
$$\begin{aligned} & R_{t+1} + \gamma Q(S_{t+1}, A') && (12.8) \\ =& R_{t+1} + \gamma Q \left(S_{t+1}, \underset{a'}{\operatorname{argmax}}\, Q(S_{t-1}, a') \right) && (12.9) \\ =& R_{t+1} + \gamma \max_{a' \in \mathcal{A}} Q(S_{t+1}, a') && (12.10) \end{aligned}$$

This means the update can be written:

$$Q(S_t, A_t) \mapsto Q(S_t, A_t) + \alpha \left(R_{t+1} + \gamma \max_{a' \in \mathcal{A}} Q(S_{t+1}, a') - Q(S_t, A_t) \right) \tag{12.11}$$

This is also known as the *sarsa-max* algorithm, because of the parallels to the sarsa algorithm.

It can be shown, that this strategy converges to an optimal policy.



	Full Backup (DP)	*Sample Backup (TD)*
Bellman Expectation Equation for $v_\pi(s)$	Iterative Policy Evaluation	TD Learning
Bellman Expectation Equation for $q_\pi(s,a)$	Q-Policy Iteration	Sarsa
Bellman Optimality Equation for $q_*(s,a)$	Q-Value Iteration	Q-Learning

$v_\pi(s) \leftarrow s$
a
r
$v_\pi(s') \leftarrow s'$
$q_\pi(s,a) \leftarrow s,a$
r
s'
$q_\pi(s',a') \leftarrow a'$
S,A
R
S'
A'
$q_*(s,a) \leftarrow s,a$
r
s'
$q_*(s',a') \leftarrow a'$

Figure 12: Dynamic programming and temporal-difference learning applied to solving various Bellman equations.

12.5 Overview: DP vs. TD learning

As we've previously discussed, dynamic programming uses full backups, while temporal-difference learning uses sample backups.

Figure 12 shows how each of these methods approach solving different form of Bellman equations. All of these algorithms we have encountered before. 13 shows the corresponding update formulas.

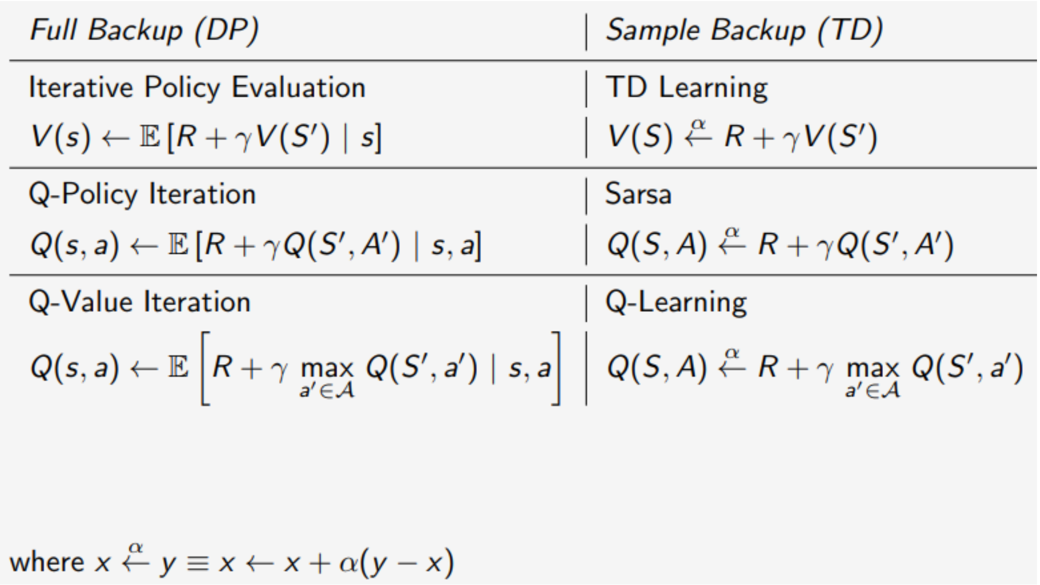
13 Value function approximation

The methods presented so far suffer from the problem, that it is hard to handle cases where the state and/or action space is very large, or even infinite. So far, we have been thinking of $v(s)$ and $q(s,a)$ essentially as look-up tables, but such a table would often be too big to fit into any reasonable computer memory.

A workaround is to use some kind of *function approximation* to the value functions. Such functions will have a set of parameters, simply summarised as $\mathbf{w}$:

$$\hat{v}(s,\mathbf{w}) \approx v(s), \quad \hat{q}(s,a,\mathbf{w}) \approx q(s,a) \tag{13.1}$$

The job of such approximator functions is to generalize from seen states to



Full Backup (DP)	*Sample Backup (TD)*
Iterative Policy Evaluation $V(s) \leftarrow \mathbb{E}[R + \gamma V(S') \mid s]$	TD Learning $V(S) \xleftarrow{\alpha} R + \gamma V(S')$
Q-Policy Iteration $Q(s, a) \leftarrow \mathbb{E}[R + \gamma Q(S', A') \mid s, a]$	Sarsa $Q(S, A) \xleftarrow{\alpha} R + \gamma Q(S', A')$
Q-Value Iteration $Q(s, a) \leftarrow \mathbb{E}\left[R + \gamma \max_{a' \in \mathcal{A}} Q(S', a') \mid s, a\right]$	Q-Learning $Q(S, A) \xleftarrow{\alpha} R + \gamma \max_{a' \in \mathcal{A}} Q(S', a')$

where $x \xleftarrow{\alpha} y \equiv x \leftarrow x + \alpha(y - x)$

Figure 13: Update formulas corresponding to figure 12.

unseen states.

We can use the strategies we already know from MC and TD learning, but instead of updating a table of values, we update the parameters $\mathbf{w}$.

13.1 Types of approximators

Figure 14 shows three types of such approximators:

- A state-value function approximator: Input a state s and get out an approximate state-value function $\hat{v}(s, \mathbf{w})$.

- An action-value function approximator where both state s and action a are input, to give an approximate action-value function $\hat{q}(s, a, \mathbf{w})$. This is known as *action in*.

- An action-value function approximator where just the state s is input, and the approximate action-value function $\hat{q}(s, a, \mathbf{w})$ for all $a \in \mathcal{A}$ is output. This is known as *action out*.

13.2 Function types

Another good question is what form the approximator function should take. Here is a non-exhaustive list of options:

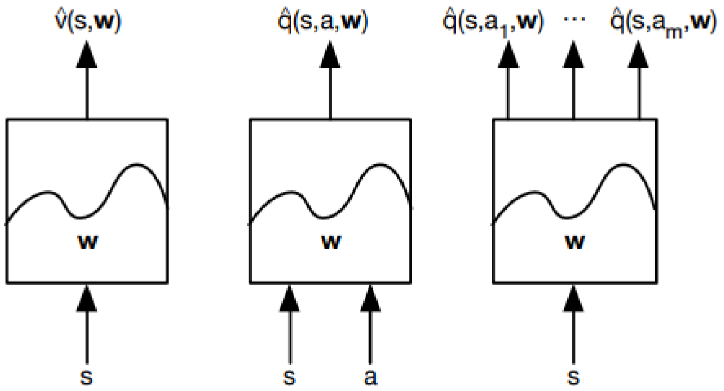


Figure 14: Three different kinds of "black box" function approximators.

- Linear combination of features.
- Neural network.
- Decision tree.
- Nearest neighbor.
- Wavelet bases.

Of course, what will and will not work depends on the problem.

Here, we will focus on functions which are differentiable, as this allows us to use gradient methods. In the list above, only the first two items are differentiable, and hence these are the ones this section will focus on.

Also, since reinforcement learning differs from supervised learning in the sense that our data points will not be stationary and/or iid., we require suitable training methods which reflect this.

13.3 Gradient descent

Gradient descent is a method for finding the minimum of a function $J(\mathbf{w})$, where $\mathbf{w}$ is an n-dimensional parameter vector. To do this, we find the gradient:

$$\nabla_{\mathbf{w}} J(\mathbf{w}) = \begin{pmatrix} \frac{\partial J}{\partial w_1} \\ \vdots \\ \frac{\partial J}{\partial w_n} \end{pmatrix} \tag{13.2}$$

We may then search for a minimum greedily, by nudging the current $\mathbf{w}$ against the direction of the gradient:

$$\mathbf{w} \mapsto \mathbf{w} - \frac{1}{2}\alpha\nabla_{\mathbf{w}}J(\mathbf{w}) \tag{13.3}$$

Here α is called the *learning rate*. The factor of 1/2 is for mathematical convenience (some presentations omit this).

There is of course no guarantee that gradient descent will not leave us stuck at a local minimum.

13.4 Gradient descent for known state-value function

Imagine (not very realistic) situation where all state-value functions for a policy π are already know - we can think of them as given to us by some kind of oracle. In this case, we would like our approximator function $\hat{v}(s, \mathbf{w})$ to minimize the following quantity, known as the *objective function*:

$$J(\mathbf{w}) = \mathbb{E}_\pi[(v(s) - \hat{v}(s, \mathbf{w}))^2] \tag{13.4}$$

If we apply gradient descent to this, each step changes $\mathbf{w}$ by:

$$\Delta\mathbf{w} = -\frac{1}{2}\alpha\nabla_{\mathbf{w}}J(\mathbf{w}) = \alpha\mathbb{E}_\pi[(v(s) - \hat{v}(s, \mathbf{w}))\nabla_{\mathbf{w}}\hat{v}(s, \mathbf{w})] \tag{13.5}$$

Here, the convenience of the factor of 1/2 becomes clear. This is known as *full gradient descent* (corresponding to batch gradient descent for supervised learning).

The above uses expectation over all possible traces through the state space. *Stochastic gradient descent* instead update after each individual step:. i.e. by sampling

$$\Delta\mathbf{w} = \frac{1}{2}\alpha\nabla_{\mathbf{w}}J(\mathbf{w}) = \alpha \underbrace{(v(s) - \hat{v}(s, \mathbf{w}))} \nabla_{\mathbf{w}}\hat{v}(s, \mathbf{w}) \tag{13.6}$$

The underbraced part can be thought of as an error term: It tells how far off from the true value we currently are.

13.5 Linear combination of features

A *feature* x is a function of the state $x = x(s)$. It may be a quantity that a domain expert has judged to be of imporance for the problem. Or it may be by necessity if the whole state s is not available to us through the environment-agent interaction. At any rate, good features throw away information in a smart way, so that a state may be more compactly represented.

A neural network has the advantage that it essentially finds useful features as part of the training process.

Imagine we describe the state s by an n-dimensional feature vector $\mathbf{x}$:

$$\mathbf{x}(s) = \begin{pmatrix} x_1(s) \\ \vdots \\ x_n(s) \end{pmatrix} \tag{13.7}$$

Then a simple function approximator is simply a linear combination of these:

$$\hat{v}(s, \mathbf{w}) = \mathbf{x}(s)^t \mathbf{w} \tag{13.8}$$

In this case, the objective function is convex, which means that there is only one local minimum, so gradient descent should work well. The gradient is also very simple:

$$\nabla_{\mathbf{w}} \hat{v}(s, \mathbf{w}) = \mathbf{x}(s) \tag{13.9}$$

This means that equation 13.6 becomes:

$$\Delta \mathbf{w} = \alpha(v(s) - \hat{v}(s, \mathbf{w}))\mathbf{x}(s) \tag{13.10}$$

I.e. the weight update is equal to learning rate times prediction error times feature value.

13.6 Gradient descent without supervision

So far we have assumed the true value function to be known, but of course in reality that's not the case. So how do we deal with this? The answer is, that we can take the same approach as we did for MC and TD-learning: Instead of using the true (unknown) values, we substitue some estimate - a target - of the value function.

13.6.1 Monte Carlo learning

For Monte Carlo learning, the target is G_t - the discounted reward for the entire rest of the episode - is an unbiased, if noisy sample of the true $v_\pi(s)$:

$$\Delta \mathbf{w} = \alpha(G_t - \hat{v}(S_t, \mathbf{w}))\nabla_{\mathbf{w}} \hat{v}(S_t, \mathbf{w}) \tag{13.11}$$

We may think of this as gathering a set of ”training data” consisting of the pairs:

$$\langle S_1, G_1 \rangle, \langle S_2, G_2 \rangle, \cdots, \langle S_T, G_T \rangle \tag{13.12}$$

13.6.2 TD(0) learning

For TD(0), the target is $R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w})$. This is a biased estimate, but we may use it anyway:

$$\Delta \mathbf{w} = \alpha(R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w}))\nabla_{\mathbf{w}} \hat{v}(S_t, \mathbf{w}) \tag{13.13}$$

Again, we can see this as gathering a "training set":

$$\langle S_1, R_2 + \gamma \hat{v}(S_2)\rangle, \langle S_2, R_3 + \gamma \hat{v}(S_3)\rangle, \cdots, \langle S_{T-1}, R_T\rangle \tag{13.14}$$

13.6.3 TD(λ) learning

For TD(λ), the target is G_t^λ:

$$\Delta \mathbf{w} = \alpha(G^\lambda - \hat{v}(S_t, \mathbf{w}))\nabla_{\mathbf{w}} \hat{v}(S_t, \mathbf{w}) \tag{13.15}$$

The "training set" in this case is:

$$\langle S_1, G_1^\lambda\rangle, \langle S_2, G_2^\lambda\rangle, \cdots, \langle S_{T-1}, G_{T-1}^\lambda\rangle \tag{13.16}$$

As before, we can also use the equivalent backward view, using eligibility traces. But here, instead of a trace for each state-action pair, there's a trace for each parameter in $\mathbf{w}$. Which means that all the traces fit into a vector of the same dimensionality as $\mathbf{w}$. They're updated as follows:

$$E_{t+1} = \gamma\lambda E_t + \nabla_{\mathbf{w}} \hat{q}(S_t, A_t, \mathbf{w}) \tag{13.17}$$

The error term is:

$$\delta_t = R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}) - \hat{q}(S_t, A_t, \mathbf{w}) \tag{13.18}$$

And the update rule:

$$\Delta \mathbf{w} = \alpha \delta_t E_t \tag{13.19}$$

13.7 Bootstrap or not?

The question now is whether we should bootstrap or not, or rather how much. In other words, which value of λ should we use? Figure 15 shows the effect of the choice of λ for a number of classic RL problems. As we can see, $\lambda = 1$, i.e. MC generally performs rather poorly. TD(0) is a bit better, but in general, there's a sweet spot somewhere in between.

So it seems that bootstrapping is a sound idea. However, there are cases where bootstrapping causes the system to "blow" up. I.e. where value functions grows exponentially. The best known example is by Baird, known as *Baird's counterexample.* Figure 16 shows in which cases convergence is theoretically guaranteed. However, often the "cross" cases work fine in practice anyway.

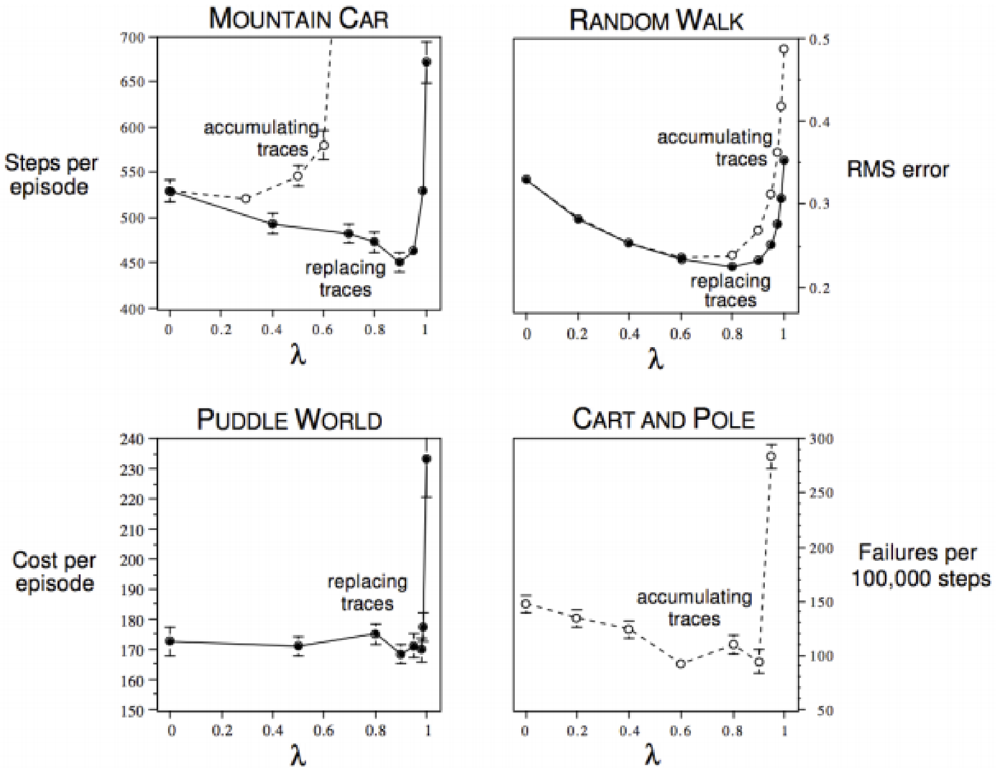
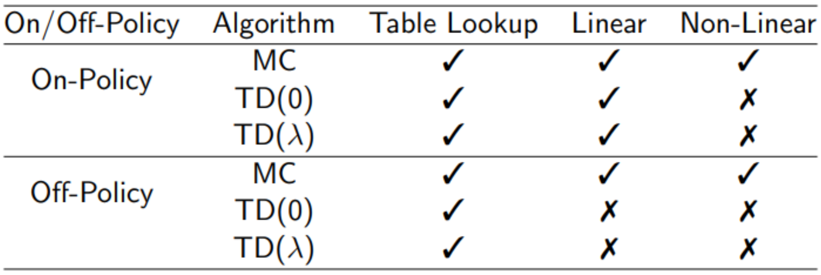


Figure 15: Error as a function of λ for four different RL problems.



On/Off-Policy	Algorithm	Table Lookup	Linear	Non-Linear
On-Policy	MC	✓	✓	✓
	TD(0)	✓	✓	✗
	TD(λ)	✓	✓	✗
Off-Policy	MC	✓	✓	✓
	TD(0)	✓	✗	✗
	TD(λ)	✓	✗	✗

Figure 16: Chart showing when convergence is guaranteed.

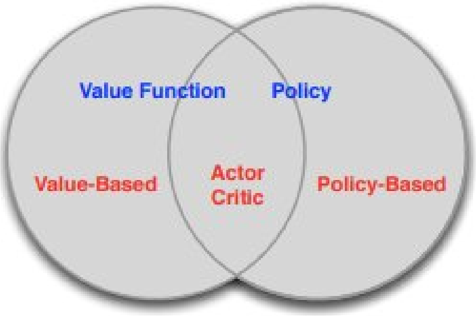


Figure 17: The relation between value-based, action-based, and actor-critical reinforcement learning.

14 Policy-based reinforcement learning

So far we have focused on using value functions to generate a policy. However, in some cases it can be useful or practical to model the policy itself directly. Again, we will consider a function approximator with a set of trainable parameters θ:

$$\pi_\theta(s, a) = \mathbb{P}[a|s, \theta] \tag{14.1}$$

As we can see, the policy is generally stochastic in nature. Also, the approach is model-free.

Pros of policy-based learning:

- Converges easily.
- Good in high dimensions and continous action spaces.
- Can learn stochastic policies.

Cons:

- Often converges to local minima.
- Policy evaluation is inefficient and high variance.