Fundamentals of Reinforcement Learning

Miguel A. Saavedra-Ruiz

February 2020

1 The K-Armed Bandit Problem

Sequential Decision Making with Evaluative Feedback

Imagine the next problem where the doctor has to give medicine to a patient and based on which one he choose, he will receive a reward (cure the patient).

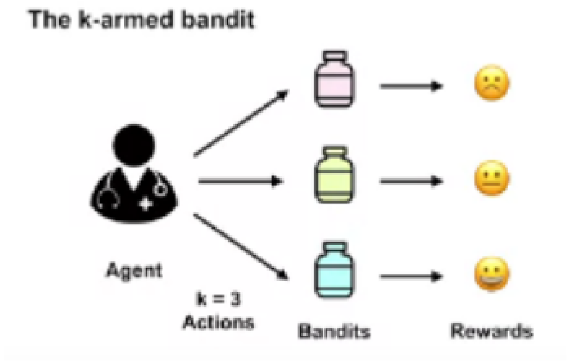


Figure 1: K-armed bandit

For the doctor to decide which action is best, we must define the value of taking each action. We call these values the **action values** or the **action value function**.

Action-Values

The value is the expected reward

$$\begin{aligned} q_*(a) &\doteq \mathbb{E}\left[R_t | A_t = a\right] \forall a \in \{1, ..., k\} \\ &= \sum_r p\left(r|a\right) r \end{aligned}$$

The goal is to maximize the expected reward

$$\underset{a}{\operatorname{argmax}}\, q_*(a)$$

The next image shows how the action values are calculated as an average. q_* is the mean of the distributions for each action.

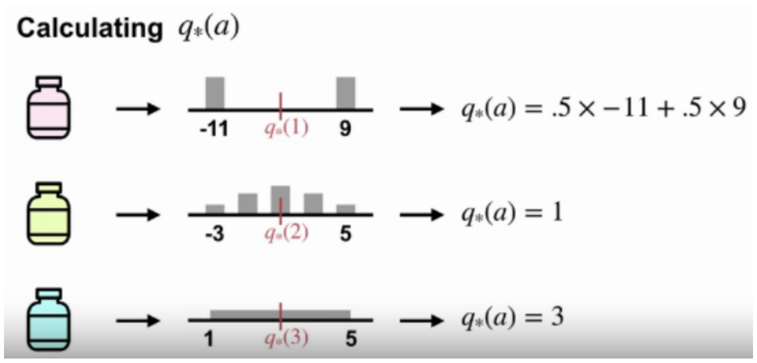


Figure 2: Calculating $q_*(a)$

Learning Action Values

The value of an action is the expected reward when the action is taken

$$q_*(a) \doteq \mathbb{E}\left[R_t | A_t = a\right]$$

$q_*(a)$ is not known, so we **estimate** it. The estimated value for action A is the **sum** of rewards observed when taking action A divided by the total number of times action A has been taken.

Sample-Average Method

The estimate of $q_*(a)$ is denoted as $Q_t(a)$ and can be written as

$$Q_t(a) \doteq \frac{\text{sum of rewards when a taken prior to t}}{\text{number of times a taken prior to t}} = \frac{\sum_{i=1}^{t-1} R_i}{t-1}$$

An example of Sample-Average methods is a Clinical Trial. A reward of 1 is obtained if the treatment succeeds otherwise 0.

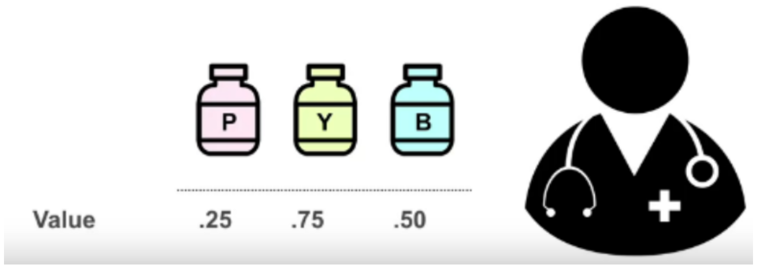


Figure 3: Action Value of Sample-Average methods example

Different attempts are made during the trial as shown in the image below. As the experiment advances, the estimates of the Action Values became more accurate Fig. 4.

In reality, our doctor would not randomly assign treatments to their patients. Instead, they would probably assign the treatment that they currently think is the best. We call this method of choosing actions **greedy**. The *greedy* action is the action that currently has the **largest estimated value**. Selecting the greedy action means the agent is **exploiting** its current knowledge. It is trying to get the **most reward** it can right now. We can compute the greedy action by taking the argmax_a of our estimated values.

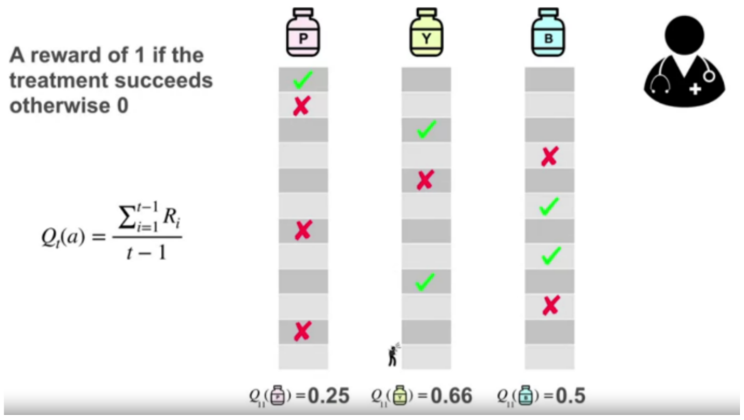

A reward of 1 if the treatment succeeds otherwise 0
$Q_t(a) = \frac{\sum_{i=1}^{t-1} R_i}{t-1}$
P
Y
B
=0.25
=0.66
=0.5

Figure 4: Sample-Average example result

An example of how is the action selection process and what corresponds to a greedy action is shown below Fig. 5.

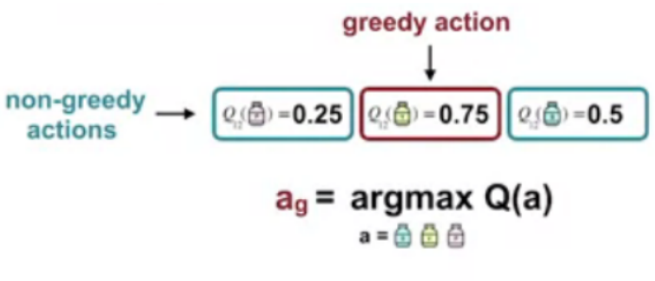


Figure 5: Greedy action example

The agent can not choose to both explore and exploit at the same time. This is one of the fundamental problems in reinforced learning. **The exploration-exploitation** dilemma.

Estimating Action Values Incrementally

The incremental update rule is derives as follows

Recall:

$$Q_n = \frac{1}{n-1} \sum_{i=1}^{n-1} R_i$$

$$Q_{n+1} = \frac{1}{n} \sum_{i=1}^{n} R_i = \frac{1}{n} \left(R_n + \sum_{i=1}^{n-1} R_i \right) = \frac{1}{n} \left(R_n + (n-1) \frac{1}{n-1} \sum_{i=1}^{n-1} R_i \right)$$

$$Q_{n+1} = \frac{1}{n} \left(R_n + (n-1) Q_n \right) = \frac{1}{n} \left(R_n + nQ_n - Q_n \right) = Q_n + \frac{1}{n} \left[R_n - Q_n \right]$$

The error in the estimate is the difference between the old estimate Q_n and the new target R_n. Taking a step towards that new target will create a new estimate that reduces our error. Here, the new **reward** is our **target**. The size of the step is determined by our step size parameter and the error of our old estimate.

Incremental update rule

$$NewEstimate \longleftarrow OldEstimate + Stepsize\,[Target - OldEstimate]$$

$$Q_{n+1} = Q_n + \alpha_n\,[R_n - Q_n]$$

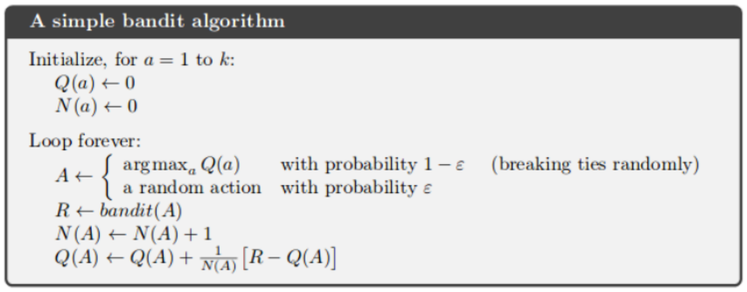
The step size α can be a function of n that produces a number from zero to one.

$$\alpha_n \to [0, 1]$$

In the specific case of the sample average, the step size is equal to one over n

$$\alpha_n = \frac{1}{n}$$

This is an example of a non-stationary bandit problem, A simple pseudo-code for this algorithm can be seen in Fig. 6 where $\frac{1}{N(a)}$ can be replaces by α and the function bandits(a) is assumed to take an action and return a corresponding reward. These problems are like the bandit problems we've discussed before, except the distribution of rewards changes with time. The doctor is unaware of this change but would like to adapt to it.



```
A simple bandit algorithm

Initialize, for $a = 1$ to $k$:
    $Q(a) \leftarrow 0$
    $N(a) \leftarrow 0$

Loop forever:
    $A \leftarrow \begin{cases} \arg\max_a Q(a) & \text{with probability } 1-\varepsilon \quad \text{(breaking ties randomly)} \\ \text{a random action} & \text{with probability } \varepsilon \end{cases}$
    $R \leftarrow bandit(A)$
    $N(A) \leftarrow N(A) + 1$
    $Q(A) \leftarrow Q(A) + \frac{1}{N(A)}[R - Q(A)]$
```

Figure 6: Bandits Pseudo-code

One option is to use a fixed step size. If α_n is constant like 0.1, then the most recent rewards affect the estimate more than older rewards Fig. 7.

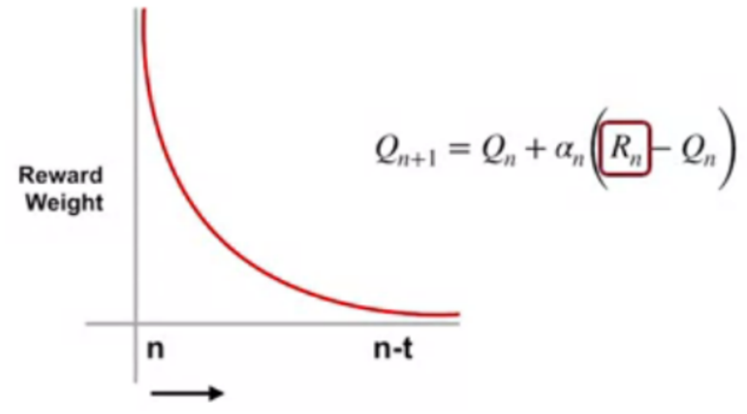


Figure 7: Non-Stationary bandit

Decaying past rewards

The prove of the efectiveness of α decaying the past rewards is shown as follows.

$$Q_{n+1} = Q_n + \alpha_n \left[R_n - Q_n\right]$$

$$Q_{n+1} = \alpha R_n + (1-\alpha)\,\alpha R_{n-1} + (1-\alpha)^2\,\alpha R_{n-2} + \cdots + (1-\alpha)^{n-1}\,\alpha R_1 + (1-\alpha)^n\,Q_1$$

$$Q_{n+1} = (1-\alpha)^n\,Q_1 + \sum_{i=1}^{n} \alpha = (1-\alpha)^{n-i}\,R_i$$

$$Q_1 \rightarrow \text{Initial Action-Value}$$

The first term tells us that the contribution of Q_1 decreases exponentially with time. The second term tells us the rewards further back in time contribute exponentially less to the sum. Taken all together, we see that the influence of our initialization of Q goes to zero with more and more data. The most recent rewards contribute most to our current estimate.

What is the trade-off

q of a is the estimated value for picking that meal. N of a is the number of times you have picked that meal, $q_*(a)$ is the value of each meal Fig. 8.

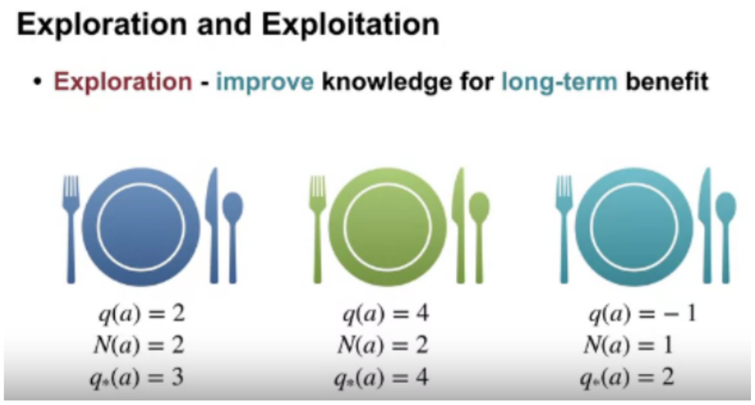

Exploration and Exploitation
• Exploration - improve knowledge for long-term benefit
q(a) = 2
N(a) = 2
q*(a) = 3
q(a) = 4
N(a) = 2
q*(a) = 4
q(a) = − 1
N(a) = 1
q*(a) = 2

Figure 8: Exploration

Exploration improves the knowledge for the long-term benefit. **Exploitation** on the other hand, exploits the agent's current estimated values. It chooses the greedy action to try to get the most reward. But by being greedy with respect to estimated values, may not actually get the most reward Fig. 9.

Epsilon-Greedy Action Selection

The ϵ-greedy selecion process is given as follows

$$A_t \leftarrow \begin{cases} \operatorname{argmax}_a Q_t(a), & \text{with probability } 1-\epsilon \\ a \sim \text{Uniform}\left(\{a_1 \ldots a_k\}\right), & \text{with probability } \epsilon \end{cases}$$

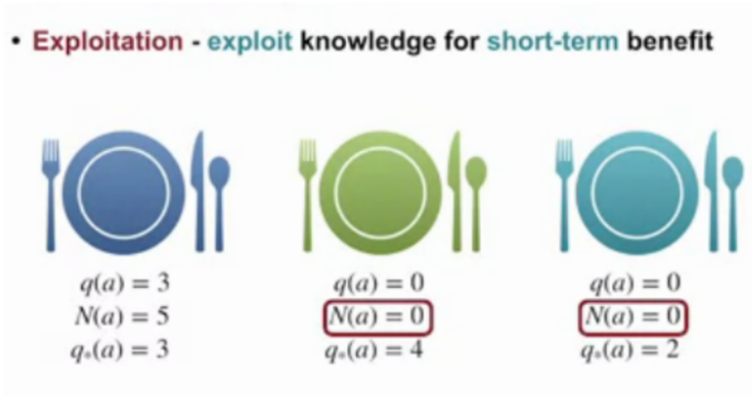


Figure 9: Exploitation

Optimistic Initial Values

Previously the initial estimated values were assumed to be 0, which is not necessarily optimistic. Now, our doctor optimistically assumes that each treatment is highly effective before running the trial. To make sure we're definitely overestimating, let's make the initial value for each action 2. Let's assume the doctor always chooses the greedy action.

The initial optimistic values are given as follows:

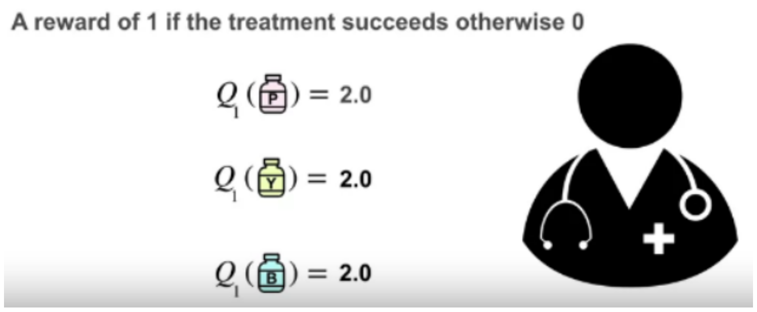


Figure 10: Optimistic values initialization

As the experiment advances, we can see how this optimistic values incentive exploration during the initial steps of the experiment, making Q closer to q_*

Some of the limitations of optimistic initial values are

- Optimistic initial values only drive early exploration
- They are not well-suited for non-stationary problems
- We may not know what the optimistic initial value should be

Upper-Confidence Bound (UCB) Action Selection

If we had a notion of uncertainty in our value estimates, we could potentially select actions in a more intelligent way. What does it mean to have uncertainty in the estimates? $Q(a)$ here represents our current estimate for action a. These brackets represent a confidence interval around $q_*(a)$. They say we are confident that the value of action a lies somewhere in this region. For instance, we believe it may be here or here.

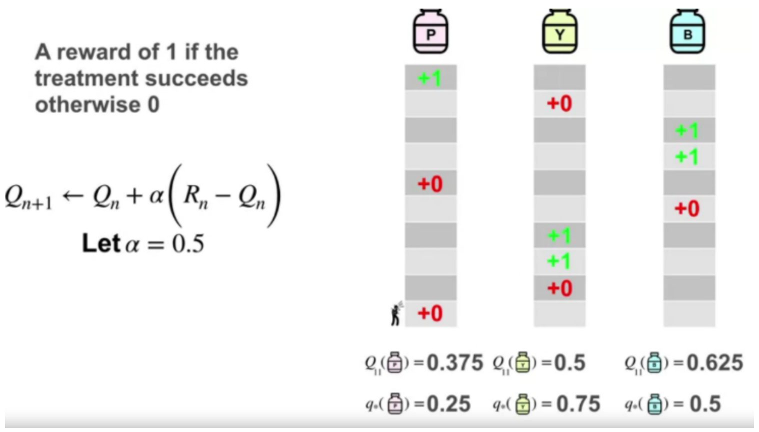


Figure 11: Optimistic values experiment

The left bracket is called the lower bound, and the right is the upper bound. The region in between is the **confidence interval** which represents our **uncertainty**. If this region is very small, we are very certain that the value of action a is near our estimated value. If the region is large, we are uncertain that the value of action A is near or estimated value.

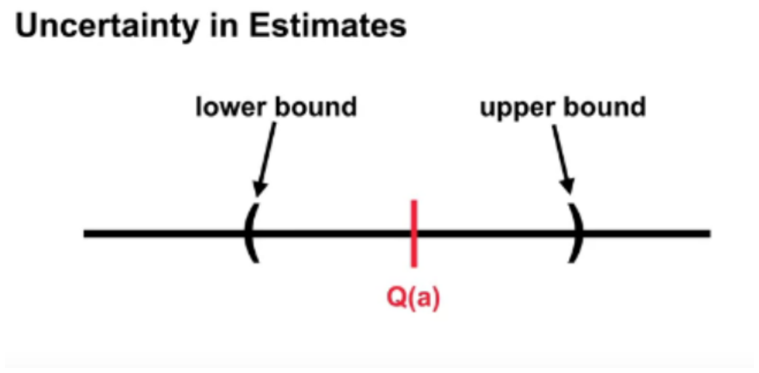


Figure 12: Uncertainty in estimates

For instance, say we have these three actions with associated uncertainties Fig. 13, our agent has no idea which is best. So it optimistically picks the action that has the highest upper bound. This makes sense because either it does have the highest value and we get good reward, or by taking it we get to learn about an action we know least about like the example on the slide.

The next equation defines the action selection process based on the Upper-Confidence Bound (UBC).

$$A_t \doteq \operatorname{argmax}\left[Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}}\right]$$

The C parameter is an user-specified parameter that controls the amount of exploration. $Q_t(a)$ represents the Exploit term and $c\sqrt{\frac{\ln t}{N_t(a)}}$ is the Exploration term.

To see the effects of this action selection criteria, Fig. 14 shows how C affects the equation.

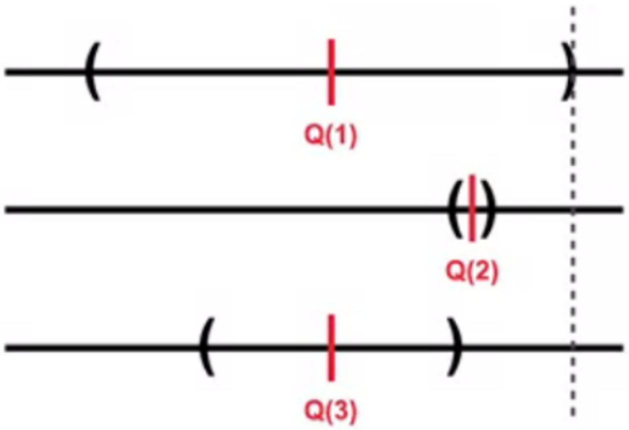


Figure 13: Uncertainty in estimates example

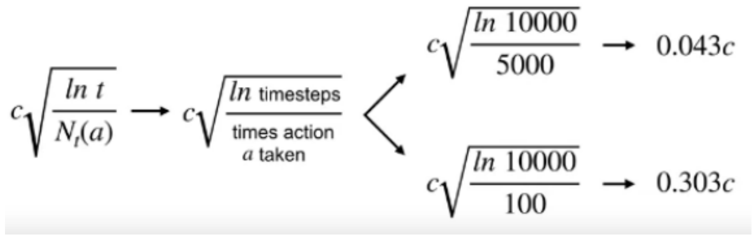

$c\sqrt{\frac{ln\ t}{N_t(a)}}$
$c\sqrt{\frac{ln\ \text{timesteps}}{\text{times action } a \text{ taken}}}$
$c\sqrt{\frac{ln\ 10000}{5000}}$
0.043c
$c\sqrt{\frac{ln\ 10000}{100}}$
0.303c

Figure 14: (UBC)

2 Markov Decision Processes

Markov Decision Processes

The k-Armed Bandit problem does not account for the fact that different situations call for different actions. It's also limited in another way. A bandit rabbit would only be concerned about immediate reward and so it would go for the carrot. But a better decision can be made by considering the long-term impact of our decisions.

Now, let's look at how the situation changes as the rabbit takes actions. We will call these situations **states**. In each state the rabbits selects an **action**. For instance, the rabbit can choose to move right. Based on this action the world changes into a new state and produces a **reward** Fig. 15. In this case, the rabbit eats the carrot and receives a reward of plus 10. However, the rabbit is now next to the tiger. Let's say the rabbit chooses the left action. The world changes into a new state or the tiger eats the rabbit and the rabbit receives a reward of minus 100.

The next diagram shows the iteration mentioned before and the relation agent - environment Fig. 16

From a set of State-Action the agent receive a Reward and ended in state S_{t+1} Fig. 17

The transition dynamics function p, formalizes this notion. Given a state S and action a, p tells us the joint probability of next state S' and reward R. We will typically assume that the set of states, actions, and rewards are finite. Since p is a probability distribution, it must be non-negative and it's sum over all possible next states and rewards must equal one. Note that future state and reward only depends on the current state and action. This is called the **Markov property**. It means that the present state is sufficient and remembering earlier states would not improve predictions about the future.

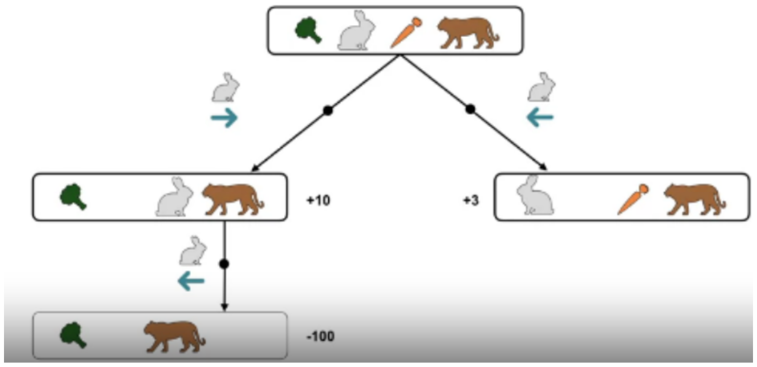


Figure 15: The rabbit example

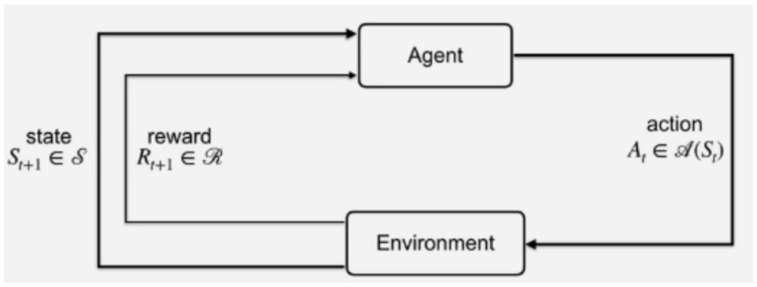

Agent
Environment
state
$S_{t+1} \in \mathcal{S}$
reward
$R_{t+1} \in \mathcal{R}$
action
$A_t \in \mathcal{A}(S_t)$

Figure 16: Relation Agent and Environment

$$p\left(s', r | s, a\right)$$

$$p : S \times R \times S \times A \rightarrow [0, 1]$$

$$\sum_{s' \in S} \sum_{r \in R} p\left(s', r | s, a\right) = 1, \forall s \in S, a \in A(s)$$

The dynamic of an MDP is shown in a diagram in Fig. 18

The Goal of Reinforcement Learning

In reinforcement learning, the agent's **objective** is to **maximize future reward**. We will formalize this notion. Perhaps we can just maximize the immediate reward as we did in bandits. Unfortunately this won't work in an MDP. An action on this time step might yield large reward because the agent of transition into a state that yields low reward. So what looked good in the short-term, might not be the best in the long-term.

The return at time step t, is simply the sum of rewards obtained after time step t. We denote the return with the letter G. The return G is a random variable because the dynamics of the MDP can be stochastic.

$$G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + \ldots$$

We maximize the expected return. For this to be well-defined, the sum of rewards must be finite. Specifically, let say there is a final time step called T where the agent environment interaction ends.

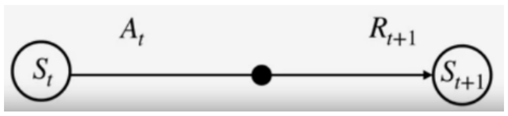


Figure 17: State-Action-Reward

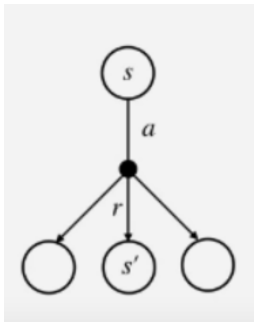


Figure 18: Dynamics of an MDP

$$\mathbb{E}\left[G_t\right] = \mathbb{E}\left[R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T\right]$$

In the simplest case, the interaction naturally breaks into chunks called **episodes**. Each episode begins independently of how the previous one ended. At termination, the agent is **reset** to a start state. Every episode has a final state which we call the terminal state. We call these tasks **episodic tasks** Fig. 19.

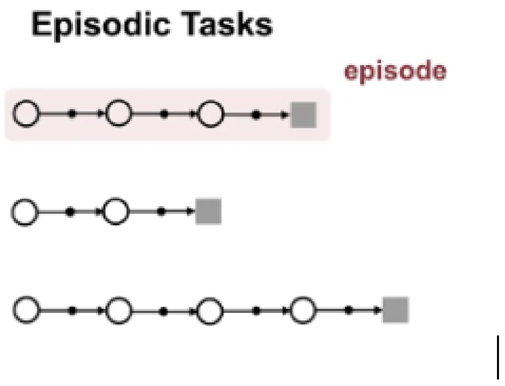


Figure 19: Episodic tasks

Goals as Rewards

There are basically two ways to set up a MDP, these are.

- 1 for goal, 0 otherwise
  - goal-reward representation
- -1 for not goal, 0 once goal reached
  - action-penalty representation

Continuing Tasks

There is a problem when implementing the Return representation of episodic tasks into continuing tasks.

Episodic Tasks

- Interaction breaks naturally into episodes
- Each episode ends in a terminal state
- Episodes are independent

$$G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + \dots + R_T$$

Continuing Tasks

- Interaction goes on actually
- No terminal state

$$G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + \dots = \infty?$$

The return formulation can then be modified to include discounting. The effect of discounting on the return is simple, immediate rewards contribute more to the sum. Rewards far into the future contribute less because they are multiplied by Gamma raised to successively larger powers of k.

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots + \gamma^{k-1} R_{t+k} + \dots$$

$$= \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

The last expression will be finite as long as $0 \leq \gamma < 1$. We have to guarantee the discounting with γ

Assume R_max is the maximum reward the agent can receive

$$G_t \doteq \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \leq \sum_{k=0}^{\infty} \gamma^k R_{max} = R_{max} \sum_{k=0}^{\infty} \gamma^k$$

Converging $\sum_{k=0}^{\infty} \gamma^k$ geometric series when $\gamma < 1$

$$G_t = R_{max} \times \frac{1}{1-\gamma}$$

The expression $R_{max} \times \frac{1}{1-\gamma}$ is finite and then G_t is finite.

Effect of γ on agent behavior

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{k-1} R_{t+k} + \dots$$

with $\gamma = 0$

$$= R_{t+1} + 0R_{t+2} + 0^2 R_{t+3} + \cdots + 0^{k-1} R_{t+k} + \dots$$

$$= R_{t+1}$$

With $\gamma = 0$ is evident that the agent only cares about the inmediate reward. This is called a **Short-sighted agent**

With $\gamma = 1$ agent takes future rewards into account more strongly. This is called a **Far-sighted agent**.

Recursive nature of returns

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \dots$$

$$G_t = R_{t+1} + \gamma \left(R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + \dots\right)$$

Recall that $\left(R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + \dots\right)$ is G_{t+1}

$$G_t = R_{t+1} + \gamma G_{t+1}$$

Remember that by definition G_{t+1} is equal to zero in the final episode

3 Value Functions & Bellman Equations

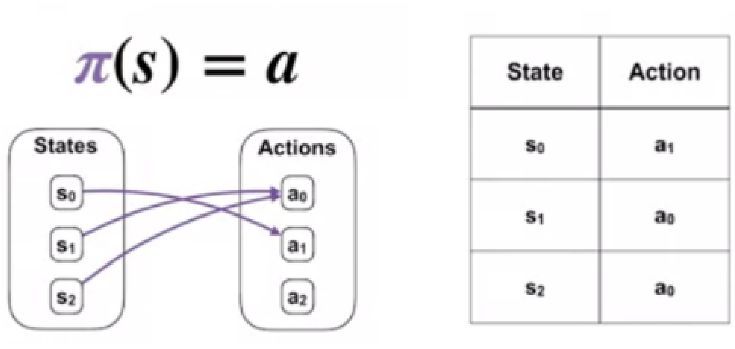
Specifying Policies

Policies are how an agent selects actions.

In the simplest case, a policy maps each state to a single action. This kind of policy is called the **deterministic policy**. We will use the fancy Greek letter π to denote a policy. $\pi(S)$ represents the action selected in state S by the policy π. In this example, π selects the action $A1$ in state S_0 and action A_0 in states S_1 and S_2. We can visualize a deterministic policy with a table. Each row describes the action chosen by π in each state. Notice that the agent can select the same action in multiple states, and some actions might not be selected in any state Fig. 20.

The arrows describe one possible policy, which moves the agent towards its house. Each arrow tells the agent which direction to move in each state. In general, a policy assigns probabilities to each action in each state Fig. 21.

In general, a policy assigns probabilities to each action in each state. We use the notation $\pi(a|s)$, to represent the probability of selecting action A in a state S. A **stochastic policy** Fig. 22 is one where multiple actions may be selected with non-zero probability. Here we show the distribution over actions for state S_0 according to π. Remember that π specifies a separate distribution over actions for each state. So we have to follow some basic rules. The sum over all action probabilities must be one for each state, and each action probability must be non-negative.



State	Action
s_0	a_1
s_1	a_0
s_2	a_0

Figure 20: Deterministic policy notation

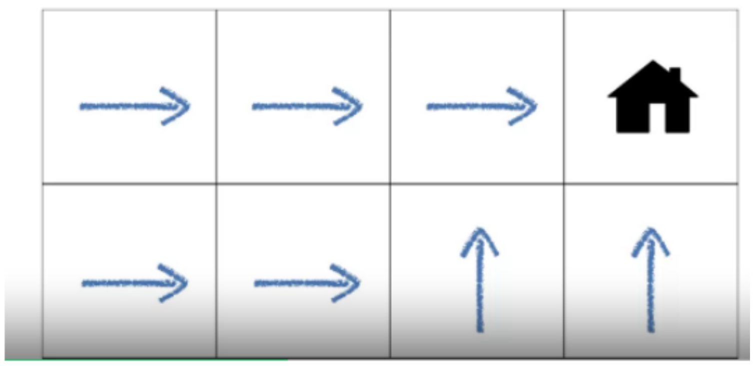


Figure 21: Deterministic policy example

A simple example of an stochastic policy can be seen in Fig. 23.

It's important that policies depend only on the current state, not on other things like time or previous states. The state defines all the information used to select the current action.

In this MDP Fig 24, we can define a policy that chooses to go either left or right with equal probability. We might also want to define a policy that chooses the opposite of what it did last, alternating between left and right actions. However, that would not be a valid policy because this is conditional on the last action. That means the action depends on something other than the state. It is better to think of this as a requirement on the state, not a limitation on the agent. In MDPs, we assume that the state includes all the information required for decision-making.

An agent's behavior is specified by a policy that maps the state to a probability distribution over actions, and two, the policy can depend only on the current state, and not other things like time or previous states.

Value Functions

The objective of RL is to learn a policy that achieves the most reward in the long run. **Value functions** formalize what this means.

Roughly speaking, a state value function is the **future award** an agent can **expect** to receive starting from a particular state. More precisely, the state value function is the **expected return from a given state**. The agent's behavior will also determine how much total reward it can expect. The definition of state-value functions can be seen in the next equation

Recall that $G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$.

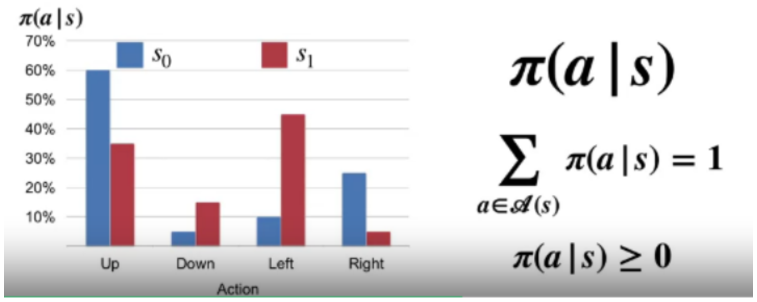


Figure 22: Stochastic policy notation

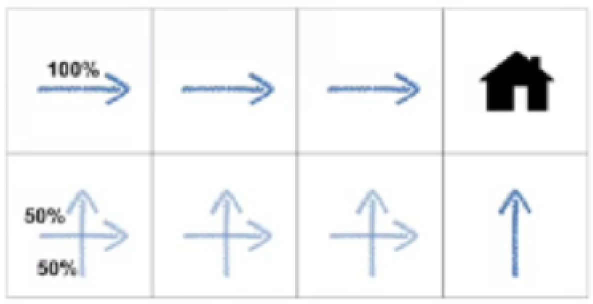


Figure 23: Stochastic policy example

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

Value function is defined with respect to a given policy. The subscript π indicates the value function is contingent on the agent selecting actions according to π. Likewise, a subscript π on the expectation $\mathbb{E}$ indicates that the expectation is computed with respect to the policy π.

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

An **action value** describes what happens when the agent **first selects** a particular **action**. More formally, the action value of a state is the expected return if the agent selects action A and then follows policy Pi.

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

Value functions are crucial in reinforce learning, they allow an agent to query the quality of its current situation instead of waiting to observe the long-term outcome. The benefit is twofold. First, the return is not immediately available and second, the return may be random due to stochasticity in both the policy and environment dynamics. The value function summarizes all the possible futures by averaging over returns. Ultimately, we care most about learning a good policy. Value function enable us to judge the quality of different policies. **Value functions predict rewards into the future**.

Imagine a chess game as shown in Fig. 25. The state value is equal to the expected sum of future rewards. Since the only possible non-zero reward is plus one for winning, the state value is simply the probability of winning if we follow the current policy π. In this two player game, the opponent's move is part of the state transition. For example, the environment moves both the agents piece, circled in blue, and the opponent's piece, circled in red. This puts the board into a new state, S'. Note, the value of state S' prime is lower than the value of state S. This means we are less likely to win the game from this new state assuming we

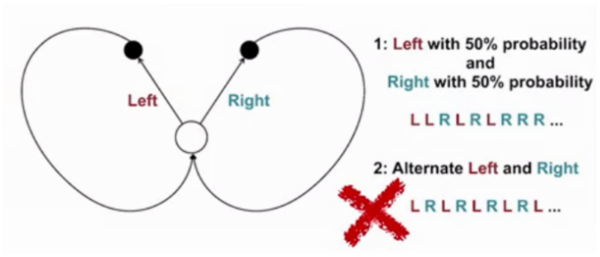

Left
Right
1: Left with 50% probability and Right with 50% probability
L L R L R L R R R ...
2: Alternate Left and Right
L R L R L R L R L ...

Figure 24: Valid and invalid policy

continue following policy π. An action value function would allow us to assess the probability of winning for each possible move given we follow the policy π for the rest of the game.

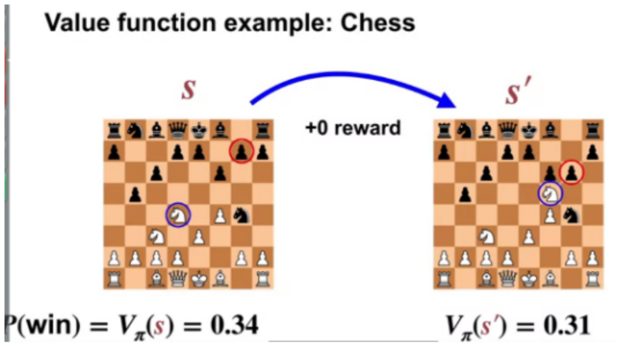
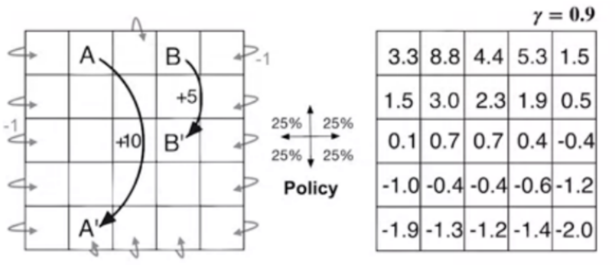


Figure 25: Value functions example in chess game

In the next example Fig. 26 the states are defined by the locations on the grid, the actions move the agent up, down, left, or right. The agent cannot move off the grid and bumping generates a reward of minus one. Most other actions yield no reward. There are two special states however, these special states are labeled A and B. Every action in state A yields plus 10 reward and plus five reward in state B. Every action in state A and B transitions the agents to states A' and B' respectively. Remember, we must specify the policy before we can figure out what the value function is. Let's look at the uniform random policy. Since this is a continuing task, we need to specify γ, let's go with 0.9. Later, we will learn several ways to compute and estimate the **value function**, but this time it will be given. On the right, we have written the value of each state.

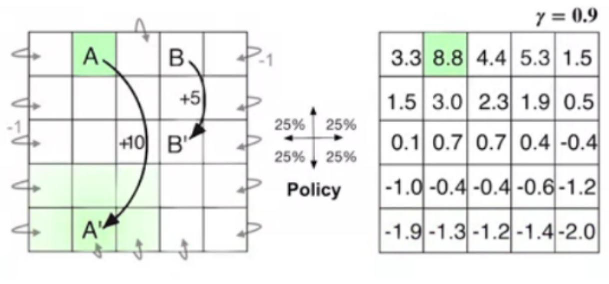


3.3	8.8	4.4	5.3	1.5
1.5	3.0	2.3	1.9	0.5
0.1	0.7	0.7	0.4	-0.4
-1.0	-0.4	-0.4	-0.6	-1.2
-1.9	-1.3	-1.2	-1.4	-2.0

Figure 26: Grid world example 1

First, notice the negative values near the bottom in Fig. 27, these values are low because the agent is likely to bump into the wall before reaching the distance states A and B. Remember, A and B are both the only sources of positive reward in this MDP. State A has the highest value, notice that the value is less than 10 even though every action from state A generates a reward of plus 10, why? Because every transition from

A moves the agent close to the lower wall and near the lower wall, the random policy is likely to bump and get negative reward.



3.3	8.8	4.4	5.3	1.5
1.5	3.0	2.3	1.9	0.5
0.1	0.7	0.7	0.4	-0.4
-1.0	-0.4	-0.4	-0.6	-1.2
-1.9	-1.3	-1.2	-1.4	-2.0

Figure 27: Grid world example 2

On the other hand, the value of state B is slightly greater than five. The transition from B moves the agent to the middle. In the middle, the agent is unlikely to bump and is close to the high-valued states A and B.

State value function refers to the expected return from a given state under a specific policy, and an **action value function** refers to the expected return from a given state after selecting a particular action and then following a given policy.

Bellman Equation Derivation

In reinforce learning **Bellman equations** allows us to relate the value of the current state to the value of future states without waiting to observe all the future rewards. We use Bellman equations to formalize this connection between the value of a state and its possible successors.

The Bellman equation for the state value function defines a relationship between the value of a state and the value of his possible successor states. Now, lets derive this relationship from the definitions of the state value function and return. Let's start by recalling that the state value function is defined as the expected return starting from the state S. Recall that the return is defined as the discounted sum of future rewards $G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$.

First, we expand the expected return as a sum over possible action choices made by the agent. Second, we expand over possible rewards and next states condition on state S and action a. We can break it down in this order because the action choice depends only on the current state, while the next state and reward depend only on the current state and action. The result is a weighted sum of terms consisting of immediate reward plus expected future returns from the next state S'. Note that R_{t+1} is a random variable, while the little r represents each possible reward outcome.

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

$$= \mathbb{E}_\pi \left[R_{t+1} + \gamma G_{t+1} | S_t = s \right]$$

$$= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a) \left[r + \gamma \mathbb{E}_\pi \left[G_{t+1} | S_{t+1} = s' \right] \right]$$

We can notice that this expected return is also the definition of the value function for state S'. The only difference is that the time index is $t + 1$ instead of t. This is not an issue because neither the policy nor p depends on time. Making this replacement, we get the Bellman equation for the state value function. The magic of value functions is that we can use them as a stand-in for the average of an infinite number of possible futures.

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a) \left[r + \gamma v_\pi(s')\right]$$

In the case for **Value Action Functions**, the equation does not begin with the policy selecting an action. This is because the action is already fixed as part of the state action pair. Instead, we skip directly to the dynamics function p to select the immediate reward and next state S'. Again, we have a weighted sum over terms consisting of immediate reward plus expected future return given a specific next state s'. However, unlike the Bellman equation for the state value function, we can't stop here. We want to recursive equation for the value of one state action pair in terms of the next state action pair. At the moment, we have the expected return given only the next state. To change this, we can express the expected return from the next state as a sum of the agents possible action choices. In particular, we can change the expectation to be conditioned on both the next state and the next action and then sum over all possible actions. Each term is weighted by the probability under π of selecting A' in the state S'. Now, this expected return is the same as the definition of the action value function for S' and A'. Making this replacement, we get the Bellman equation for the **action value function**.

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

$$= \sum_{s'} \sum_r p(s', r|s, a) \left[r + \gamma \mathbb{E}_\pi \left[G_{t+1} | S_{t+1} = s'\right]\right]$$

$$= \sum_{s'} \sum_r p(s', r|s, a) \left[r + \gamma \sum_{a'} \pi(a', s') \mathbb{E}_\pi \left[G_{t+1} | S_{t+1} = s', A_{t+1} = a'\right]\right]$$

$$q_\pi(s, a) = \sum_{s'} \sum_r p(s', r|s, a) \left[r + \gamma \sum_{a'} \pi(a', s') q_\pi(s', a')\right]$$

These equations provide relationships between the values of a state or state action pair and the possible next states or next state action pairs.

Why Bellman Equations?

To illustrate the power of Bellman equations, consider this small example fig. 28, consisting of just four states, labeled A, B, C and D on a grid. The action space consists of moving up, down, left and right. Actions which would move off the grid, instead keep the agent in place. Say for example we start in state C, moving up would take us to state A. If we then try to move left we would hit a wall and stay in state A. Moving right next would take us to state B. From there, moving down would land us in state D. The reward is 0 everywhere except for any time the agent lands in state B. If the agent lands in state B, it gets a reward of +5. This includes starting in state B and hitting a wall to remain there. Let's consider the uniform random policy, which moves in every direction 25% of the time. The discount factor γ 0.7.

Recall that the value function is defined as the expected return under policy π. This is an average over the return obtained by each sequence of actions an agent could possibly choose, infinitely, many possible features.

$$v_\pi(A) \doteq \mathbb{E}_\pi \left[G_t | S_t = A\right]$$

$$v_\pi(B) \doteq \mathbb{E}_\pi \left[G_t | S_t = B\right]$$

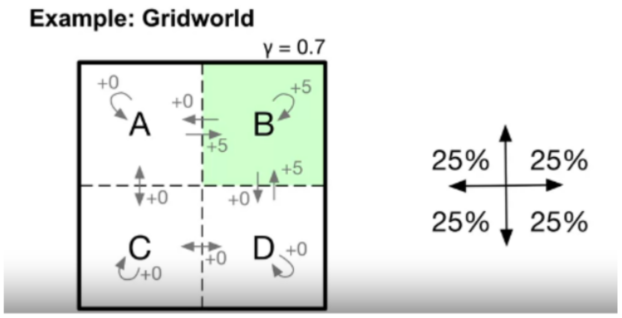


Figure 28: Square gridworld example

$$v_\pi(C) \doteq \mathbb{E}_\pi\left[G_t | S_t = C\right]$$

$$v_\pi(D) \doteq \mathbb{E}_\pi\left[G_t | S_t = D\right]$$

Using the Bellman equation, we can write down an expression for the value of state A in terms of the sum of the four possible actions and the resulting possible successor states. We can simplify the expression further in this case, because for each action there's only one possible associated next state and reward. That's the sum over S' and r reduces to a single value. Note that here s' and r do still depend on the selected action, and the current state s. But to keep the notation short, we haven't this explicitly.

If we go right from state A, we land in state B), and receive reward of +5. This happens one quarter of the time under the random policy. If we go down, we land in state C, and receive no immediate reward. Again, this occurs one-quarter of the time. If you go either up or left, we will land back in state A again. Each of the actions, up and left, again, occur one-quarter of the time. Since they both land in state A and received no reward, we combine them into a single term with factor of 1 over 2. This can be seen in Fig. 29

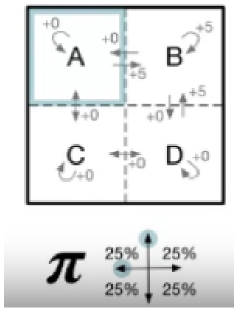


Figure 29: Square gridworld example solving state A

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)\left[r + \gamma v_\pi(s')\right]$$

$$v_\pi(A) = \sum_a \pi(a|A)\left(r + 0.7 v_\pi(s')\right)$$

$$v_\pi(A) = \frac{1}{4}\left(5 + 0.7 V_\pi(B)\right) + \frac{1}{4} 0.7 V_\pi(C) + \frac{1}{2} 0.7 V_\pi(A)$$

We can write down a similar equation for each of the other states, B, C, and D. Now we have a system of four equations for four variables. You can solve this by hand if you want, or put it into an automatic equation solver.

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a) \left[r + \gamma v_\pi(s')\right]$$

$$v_\pi(A) = \frac{1}{4}\left(5 + 0.7V_\pi(B)\right) + \frac{1}{4}0.7V_\pi(C) + \frac{1}{2}0.7V_\pi(A)$$

$$v_\pi(B) = \frac{1}{2}\left(5 + 0.7V_\pi(B)\right) + \frac{1}{4}0.7V_\pi(A) + \frac{1}{4}0.7V_\pi(D)$$

$$v_\pi(C) = \frac{1}{4}0.7V_\pi(A) + \frac{1}{4}0.7V_\pi(D) + \frac{1}{2}0.7V_\pi(C)$$

$$v_\pi(D) = \frac{1}{4}\left(5 + 0.7V_\pi(B)\right) + \frac{1}{4}0.7V_\pi(C) + \frac{1}{2}0.7V_\pi(D)$$

Solving the equations give the next results

$$v_\pi(A) = 4.2$$

$$v_\pi(B) = 6.1$$

$$v_\pi(C) = 2.2$$

$$v_\pi(D) = 4.2$$

This approach may be possible for **MDPs** of **moderate size** Fig 30. However, in more complex problems, this won't always be practical. Consider the game of chess for example. We probably won't be able to even list all the possible states, there are around 1045 of them. Constructing and solving the resulting system of Bellman equations would be a whole other story.

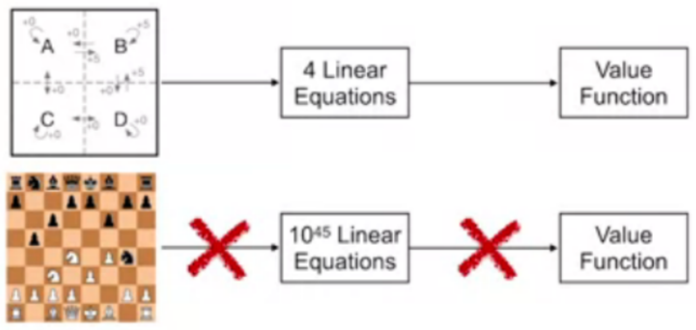


Figure 30: Limitation of MDPs

Optimal Policies

The policy π specifies how an agent behaves. Given this way of behaving, we then aim to find the value function. But the goal of reinforcement learning is not just to evaluate specific policies. Ultimately, we want to find a policy that obtains as much reward as possible in the long run.

We will say policy π_1 is **as good** as or better than policy π_2, if and only if the **value** under π_1 is **greater** than or equal to the value under π_2 **for every state** Fig. 31.

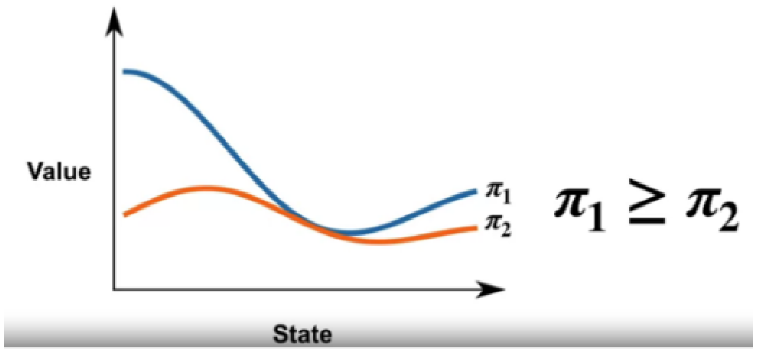


Figure 31: Optimal policies

That is, an optimal policy will have the highest possible value in every state. **There's always at least one optimal policy**, but there may be more than one. We'll use the notation π_* to denote any optimal policy Fig. 32.

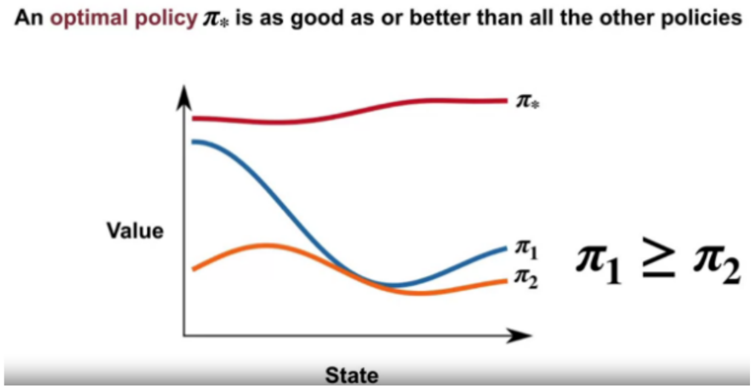


Figure 32: Optimal policy π_*

Let's say there is such a policy π_1, which does well in some states while policy π_2, does well and others. We could combine these policies into a third policy π_3, which always chooses actions according to whichever of policy π_1 and π_2 has the highest value in the current state. π_3 will necessarily have a value greater than or equal to both π_1 and π_2 in every state Fig. 33. So we will never have a situation we're doing well in one state require sacrificing value in another.

The optimal policy what will be the one for which the value of X is highest. The answer depends on the discount factor γ. Consider $\gamma = 0$ Fig. 34. In this case, the value is defined using only the immediate reward. The value of $V_{\pi 1}(X) = 1$, while the value under $V_{\pi 2}(X) = 0$ because the plus to reward occurs after a one-step delay, which does not affect the return when $\gamma = 0$. So in this case, π_1 is the optimal policy. What if instead $\gamma = 0.9$? In this case, the value of x under each policy is an infinite sum. π_1 receives an immediate reward of one followed by zero, and then one again, and so on. Each reward and the expression for the value of state X is discounted by some power of γ. We can express this compactly as a geometric

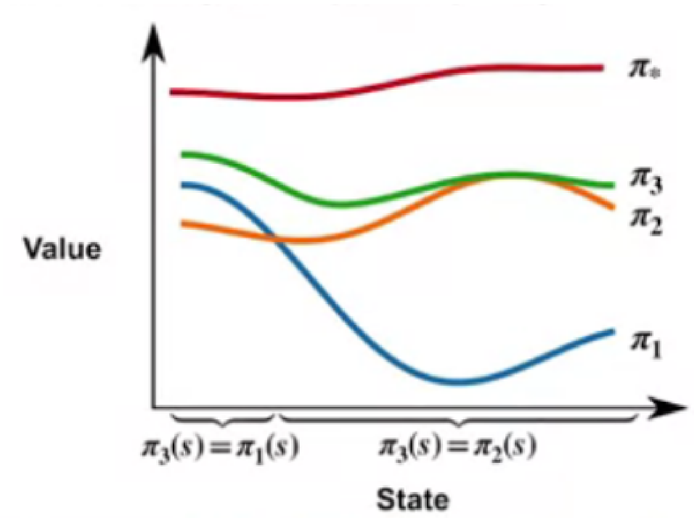


Figure 33: Optimal policy π_3

series. Applying the geometric series formula, we get the value shown here for π_1 which evaluates to around 5.3. We can write the value under π_2. Similarly, except we will receive a reward of two on every odd step instead of one on every even step. We can again write this as a geometric series and obtain a closed form solution. In this case, the solution evaluates to around 9.5.

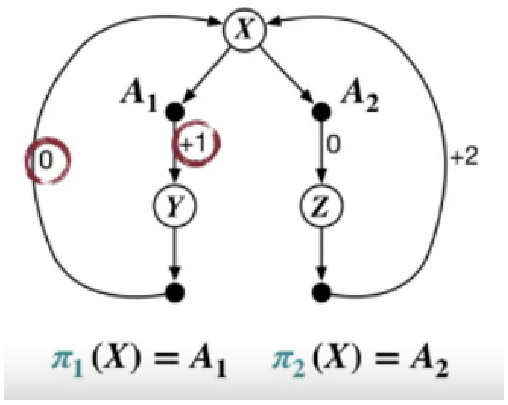


Figure 34: Optimal policy left and right example

With $\gamma = 0$

$$V_{\pi 1}(X) = 1$$

$$V_{\pi 2}(X) = 0$$

With $\gamma = 0.9$

$$V_{\pi 1}(X) = 1 + 0.9 * 0 + (0.9)^2 * 1 + \dots$$

$$V_{\pi 1}(X) = \sum_{k=0}^{\infty} (0.9)^{2k} = \frac{1}{1-0.9^2} \approx 5.3$$

$$V_{\pi 2}(X) = 0 + 0.9 * 2 + (0.9)^2 * 0 + \dots$$

$$V_{\pi 2}(X) = \sum_{k=0}^{\infty} (0.9)^{2k+1} * 2 = \frac{0.9}{1-0.9^2} * 2 \approx 9.5$$

There were only two deterministic policies, and we simply had to compute the value function for each of them. In general, it will not be so easy. Even if we limit ourselves to deterministic policies, the **number of possible policies** is equal to the **number of possible actions** to the **power of the number of states**.

Luckily, there's a better way to organize our search of the policy space. The solution will come in the form of yet another set of Bellman equations, called the Bellman's Optimally equations.

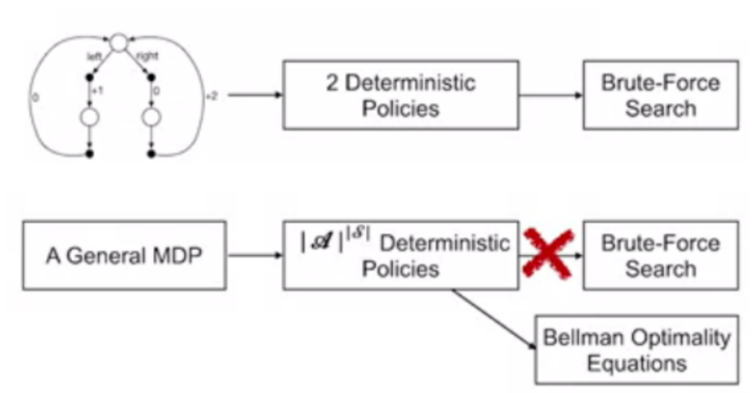


Figure 35: Why are necessary bellman optimal Equations

Optimal Value Functions

Recalling that an optimal policy is one which is as good as or better than every other policy.

The value function for the optimal policy thus has the greatest value possible in every state. We can express this mathematically, by writing that $v_{\pi *}(s)$ is **equal to the maximum value over all policies**. This holds for every state in our state-space. Taking a maximum over policies might not be intuitive. So let's take a moment to break down what it means. Imagine we were to consider every possible policy and compute each of their values for the state S. The value of an optimal policy is defined to be the largest of all the computed values. We could repeat this for every state and the value of an optimal policy would always be the largest. All optimal policies have this same optimal state-value function, which we denote by V_*. Optimal policies also share the same optimal action-value function, which is again the maximum possible for every state action pair. We denote this shared action value function by q star.

$$v_* \rightarrow v_{\pi *}(s) \doteq \mathbb{E}_{\pi *}\left[G_t | S_t = s\right] = \max_{\pi} v_{\pi}(s) \forall s \in S$$

$$q_* \rightarrow q_{\pi *}(s,a) = \max_{\pi} q_{\pi}(s,a) \forall s \in S \text{ and } a \in A$$

This is simply the Bellman equation we introduced previously for the specific case of an optimal policy. However, because this is an optimal policy, we can rewrite the equation in a special form, which doesn't reference the policy itself. Remember there always exists an optimal deterministic policy, one that selects an optimal action in every state. Such a deterministic optimal policy will assign Probability 1, for an action that achieves the highest value and Probability 0, for all other actions. We can express this another way by replacing the $\sum_a \pi_*(a|s)$ with $\max_a$. Notice that π_* no longer appears in the equation. We have derived a relationship that applies directly to v_* itself. We call this special form, the **Bellman optimally equation for v_***.

Recall that $v_\pi(s) = \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a) \left[r + \gamma v_\pi(s')\right]$

$$v_*(s) = \sum_a \pi_*(a|s) \sum_{s'} \sum_r p(s', r|s, a) \left[r + \gamma v_*(s')\right]$$

$$v_*(s) = \max_a \sum_{s'} \sum_r p(s', r|s, a) \left[r + \gamma v_*(s')\right] \leftarrow \text{Bellman Optimally Equation for } V_*$$

We can make the same replacement in the Bellman equation for the **action-value function**. Here the optimal policy appears in the inner sum. Once again, we replace the sum over Pi star with a max over a. This gives us the Bellman optimally equation for q star.

Recall that $q_\pi(s, a) = \sum_{s'} \sum_r p(s', r|s, a) \left[r + \gamma \sum_{a'} \pi(a', s') q_\pi(s', a')\right]$

$$q_*(s, a) = \sum_{s'} \sum_r p(s', r|s, a) \left[r + \gamma \sum_{a'} \pi_*(a', s') q_*(s', a')\right]$$

$$q_*(s, a) = \sum_{s'} \sum_r p(s', r|s, a) \left[r + \gamma \max_{a'} \pi_*(a', s') q_*(s', a')\right] \leftarrow \text{Bellman Optimally Equation for } q_*$$

In an earlier lecture, we discussed how the Bellman equations form a linear system of equations that can be solved by standard methods Fig. 36. The Bellman's optimally equation gives us a similar system of equations for the optimal value. One natural question is, can we solve this system in a similar way to find the optimal state-value function? Unfortunately, the answer is no. **Taking the maximum over actions is not a linear operation**. So standard techniques from **linear algebra** for solving linear systems **won't apply** Fig. 37.

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a) \left[r + \gamma v_\pi(s')\right]$$

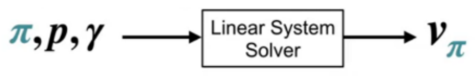


Figure 36: Bellman Linear solver for non-optimal equations

$$v_*(s) = \max_a \sum_{s'} \sum_r p(s', r|s, a) \left[r + \gamma v_*(s')\right] \rightarrow \text{the term } \max_a \text{ is not linear}$$

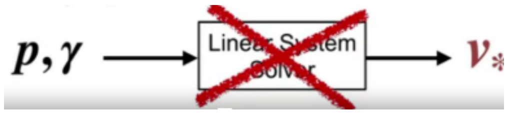


Figure 37: Bellman Linear solver for optimal equations

You might be wondering why we can't simply use π_* in the ordinary Bellman equation to get a system of linear equations for π_*. The answer is simple. **We don't know** π_*. If we did, then we would have already achieved the **fundamental goal of reinforcement learning**.

Using Optimal Value Functions to Get Optimal Policies

In general, having v_* makes it relatively easy to work out the optimal policy as long as we also have access to the dynamics function p. For any state, we can look at each available action and evaluate the boxed term. There will be some action for which this term obtains a maximum.

v_* is equal to the maximum of the boxed term over all actions. v_* is the argmax, which simply means the particular action which achieves this maximum.

$$v_*(s) = \max_a \sum_{s'} \sum_r p(s', r|s, a) \left[r + \gamma v_*(s')\right]$$

$$\pi_*(s) = \operatorname*{argmax}_a \sum_{s'} \sum_r p(s', r|s, a) \left[r + \gamma v_*(s')\right]$$

First, imagine doing so for particular action, labeled A_1 Fig. 38. We look at each state and reward which may follow from state s after taking action A_1. Since we have access to v_* and p, we can then evaluate each term in $\sum_{s'} \sum_r p(s', r|s, a) \left[r + \gamma v_*(s')\right]$. Let's say that for A_1, the boxed term evaluates to 5. We can repeat the same procedure for A_2. Again, this requires only a one step look ahead thanks to having access to v_*. Let's say in this case we find a result of 10. Finally for A_3, let's say we obtain a result of 7. Of these three actions, A_2 maximizes the boxed term with a value of 10. This means that A_2 is the optimal action.

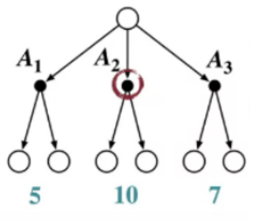
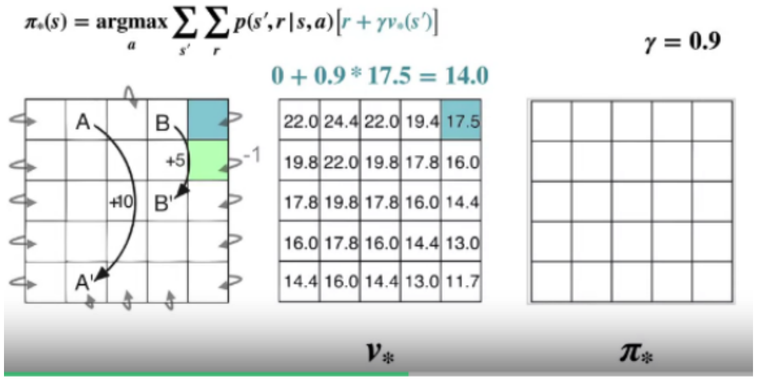


Figure 38: Optimal policy example

Imagine the next example, Imagine that the action is in the green position. The up action leads here (blue area), giving no reward and a next state value of 17.5. The sum of reward and discounted next state value is 14.0 Fig. 39.


$\pi_*(s) = \operatorname*{argmax}_a \sum_{s'} \sum_r p(s',r|s,a)[r + \gamma v_*(s')]$
$\gamma = 0.9$
$0 + 0.9 * 17.5 = 14.0$
A
B
+5
-1
+10
B'
A'
v_*
π_*

22.0	24.4	22.0	19.4	17.5
19.8	22.0	19.8	17.8	16.0
17.8	19.8	17.8	16.0	14.4
16.0	17.8	16.0	14.4	13.0
14.4	16.0	14.4	13.0	11.7

Figure 39: Optimal policy grid example 1

The right action hits the wall, giving -1 reward and leaving the agent in the same state, which has a value of 16.0. The sum of reward and discounted next state value is 13.4 Fig. 40.

The down action leads here, giving no reward, but a next state value of 14.4. After discounting, this gives 13 Fig. 41.

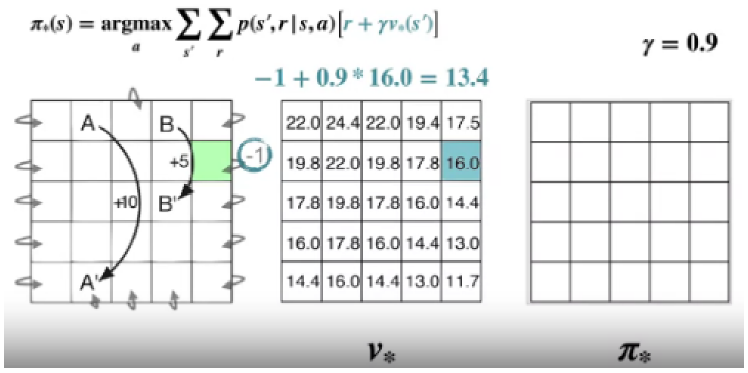


Figure 40: Optimal policy grid example 2

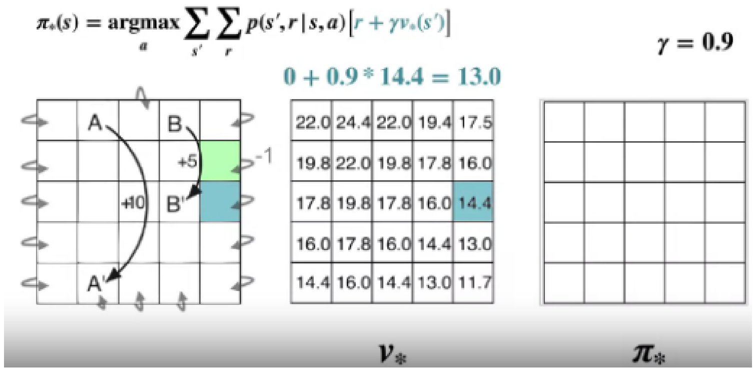


Figure 41: Optimal policy grid example 3

Finally, the left action leads here. Again, giving no reward, but a next state value of 17.8. Discounting by gamma gives us 16. Of all these choices, t**he highest value action is left at 16**. Therefore, **left is the optimal action in this state** and must be selected by any optimal policy Fig. 42.

Remember that in general, the dynamics function p can be stochastic, so it might not always be so simple. In general, the dynamics function p can be stochastic, so it might not always be so simple. However, as long as we have access to p, we can always **find the optimal** action from v_* by **computing the right-hand side of the Bellman optimally equation** for each action and finding the largest value. If instead we have access to q_*, it's even easier to come up with the optimal policy. In this case, we do not have to do a one step look ahead at all. We only have to **select any action** a**, that maximizes** $q_(s,a)$. The action-value function caches the results of a one-step look ahead for each action. In this sense, the problem of finding an optimal action-value function corresponds to the goal of finding an optimal policy.

As a reminder, the next optimal equations summarizes how to obtain π_*.

$$\pi_*(s) = \operatorname*{argmax}_a \sum_{s'} \sum_r p(s',r|s,a)\left[r + \gamma v_*(s')\right]$$

$$\pi_*(s) = \operatorname*{argmax}_a q_*(s,a)$$

The diagram of Fig. 43(a) shows how the acquisition of an optimal policy process is done. The up diagram is to obtain the optimal policy π_* from v_* and the second diagram from q_*.

On the other hand, **Policies** tell an agent how to behave in their environment. There are two king of policies: Deterministic and Stochastic. The Fig. 43(b) summarizes this information.

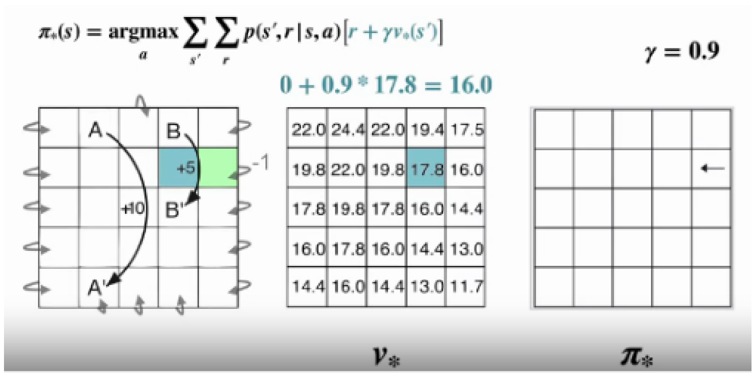
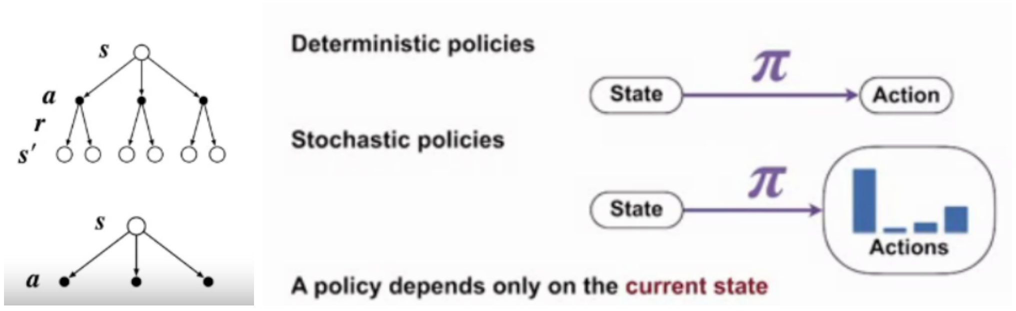


Figure 42: Optimal policy grid example 4



(a) (b)

Figure 43: (a) Optimal policy diagrams, (b) Deterministic and Stochastic policies.

4 Dynamic Programming

Policy Evaluation vs. Control

We often talk about two distinct tasks, policy evaluation and control. **Policy evaluation is the task of determining the value function for a specific policy. Control is the task of finding a policy to obtain as much reward as possible.** In other words, finding a policy which maximizes the value function. **Control is the ultimate goal of reinforcement learning**. But the task of policy evaluation is usually a necessary first step.

Dynamic programming algorithms **use** the **Bellman equations** to define iterative algorithms for both policy evaluation and control.

Imagine someone hands you a policy and your job is to determine how good that policy is. **Policy evaluation** is the task of determining the state value function v_π for a particular policy π.

Policy evaluation $\pi \rightarrow v_\pi$

Recall that $v_\pi(s) \doteq \mathbb{E}_\pi[G_t|S_t=s]$ and $G_t \doteq \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$

We have seen how the Bellman equation reduces the problem of finding v_π to a system of linear equations, one equation for each state. So the problem of policy evaluation reduces to solving this system of linear equations. In principle, we could approach this task with a variety of methods from linear algebra Fig. 44.

Recall that $v_\pi(s) = \sum_a \pi(a|s) \sum_{s'} \sum_r p(s',r|s,a)[r+\gamma v_\pi(s')]$

Control is the task of improving a policy Fig. 45. Recall that a policy π_2 is considered as good as or better than π_1 if the value under π_2 is greater than or equal to the value under π_1 in every state. We say π_2 is

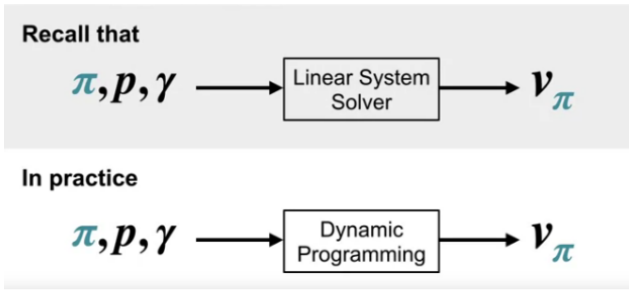


Figure 44: Dynamic programming as a set of linear equations

strictly better than π_1 if π_2 is as good as or better than π_1 and there's at least one state where the value under π_2 is strictly greater than the value under π_1.

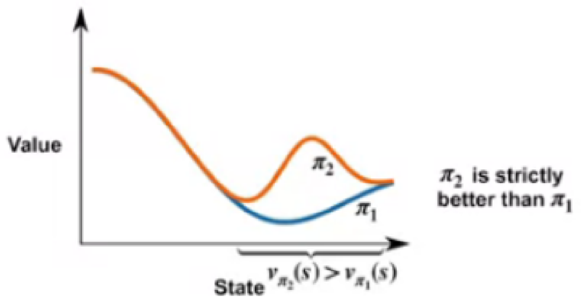


Figure 45: Control: Improving a giving policy

The goal of the control task is to modify a policy to produce a new one which is strictly better. Moreover, we can try to improve the policy repeatedly to obtain a sequence of better and better policies Fig. 46.

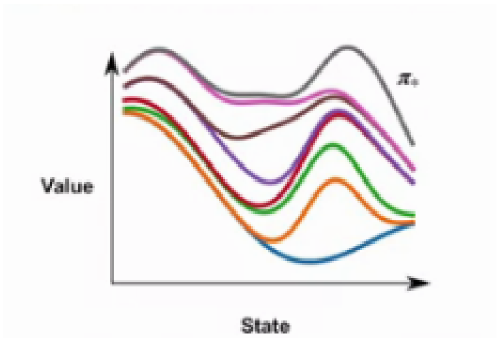


Figure 46: Improving multiple times a policy

Dynamic programming uses the various Bellman equations we've seen, along with knowledge of p, to work out value functions and optimal policies. Classical dynamic programming does not involve interaction with the environment at all. Instead, **we use dynamic programming methods to compute value functions and optimal policies given a model of the MDP**.

To summarize Fig. 47, policy evaluation is the task of determining the state value function v_π for policy π. Control is the task of improving an existing policy. And dynamic programming techniques can be used to solve both of these tasks **if we have access to the dynamics function** p.

The Pseudo code for **iterative policy evaluation** is given in Fig. 48.

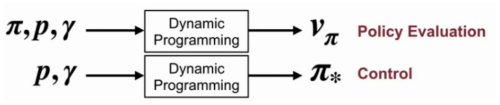
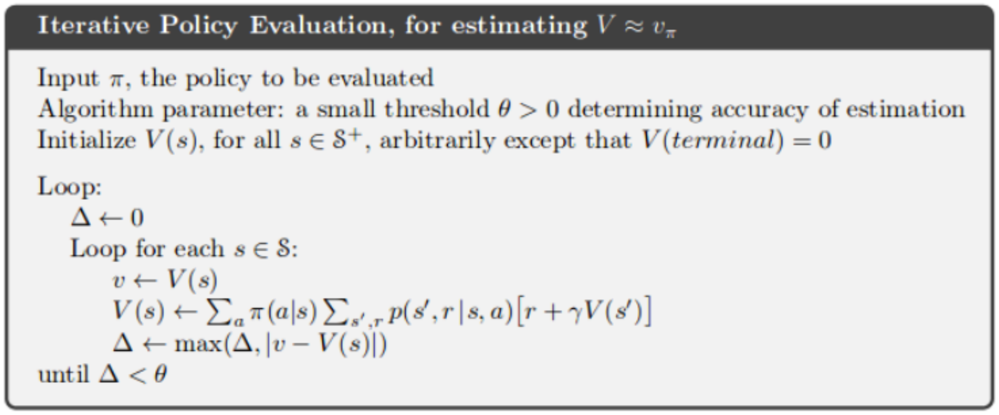


Figure 47: Dynamic programming for Evaluation and control



Iterative Policy Evaluation, for estimating $V \approx v_\pi$

```
Input $\pi$, the policy to be evaluated
Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop:
   $\Delta \leftarrow 0$
   Loop for each $s \in \mathcal{S}$:
      $v \leftarrow V(s)$
      $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)\big[r + \gamma V(s')\big]$
      $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$
```

Figure 48: Iterative policy evaluation

Iterative Policy Evaluation

The idea of **iterative policy evaluation** is so simple that at first it might seem a bit silly. We take the **Bellman equation** and directly use it **as an update rule**. Now, instead of an equation which holds for the true value function, we have a procedure we can apply to **iteratively** refine our estimate of the value function. This will produce a sequence of better and better approximations to the value function.

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)\left[r + \gamma v_\pi(s')\right]$$

$$v_{k+1}(s) \leftarrow \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)\left[r + \gamma v_k(s')\right] \leftarrow \text{Update rule}$$

We begin with an arbitrary initialization for our approximate value function, let's call this V_0. Each iteration then produces a better approximation by using the update rule shown above. Each iteration applies this updates to every state, s, in the state space, which we call a **sweep**. Applying this update repeatedly leads to a better and **better approximation** to the state value function V_π. If this update leaves the value function approximation unchanged, that is, if V_{k+1} equals V_k for all states, then V_k equals V_π, and we have found the value function. This is because V_π is the unique solution to the Bellman equation Fig. 49.

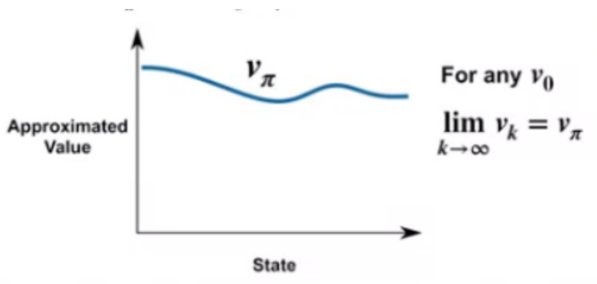
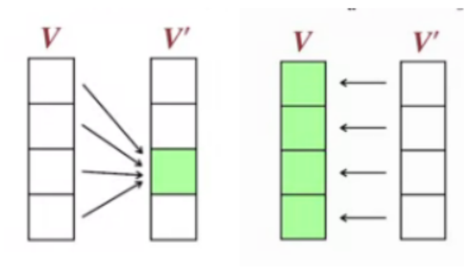


Figure 49: Approximating V_π to V_*

To implement iterative policy evaluation, we store two arrays, each has one entry for every state. One array, which we label V stores the current approximate value function. Another array, V', stores the updated values. By using two arrays, we can compute the new values from the old one state at a time without the old values being changed in the process. At the end of a full sweep, we can write all the new values into V; then we do the next iteration Fig. 50.



(a) (b)

Figure 50: (a) Updating V', (b) Updating V.

It is also possible to implement a **version with only one array**, in which case, some updates will themselves use new values instead of old. This single array version is still guaranteed to converge, and in fact, **will usually converge faster**. This is because it gets to use the updated values sooner.

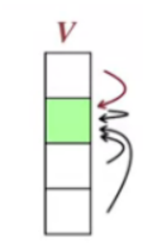


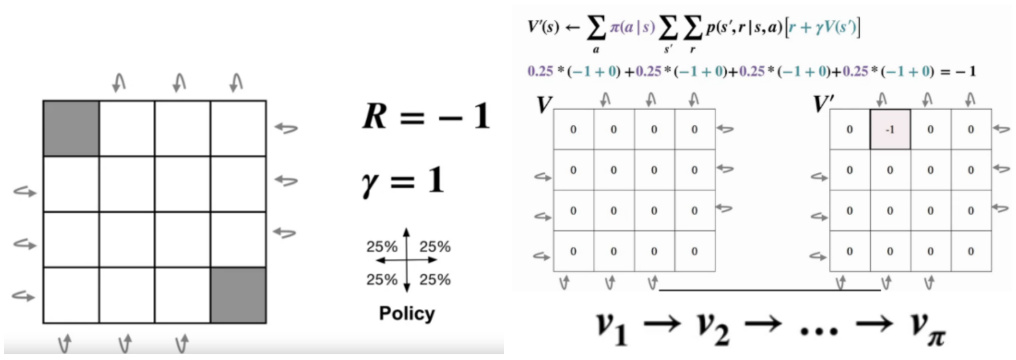
Figure 51: Policy evaluation with one array

For the next example Fig. 52(a) we initialize all the values in V to zero, all transitions leads to a reward of -1,when we gen onto one gray box the episode ends. The initial value stored in V' are relevant since they'll always be updated before they are used. We can now begin our first iteration with the update to state one. We start from the pink box in Fig. 52(b) To compute the update, we have to sum over all actions. Consider the left action first, which has probability one-quarter under the uniform random policy. The dynamics function, p, is deterministic here so only the reward and value for $V(s')$ contributes to the sum. The sum includes -1 for the reward, and zero for the value of the terminal state. Since we initialized all state values to zero, and the reward for each transition is minus one the computation for all the other actions will look much the same. The result is that v' of state one is set to minus one.

Finally, as a reminder, the pseudo-code for this algorithm is shown in Fig. 48

Policy Improvement

Previously, we showed that given v_*, we can find the optimal policy by choosing the Greedy action. The Greedy action maximizes the Bellman's optimally equation in each state. Imagine instead of the optimal value function, we select an action which is greedy with respect to the value function v_π of an arbitrary policy π. What can we say about this new policy? That it is greedy with respect to v_π. The first thing to note is that this new policy must be different than π. If this greedification doesn't change π, then π was



$R = -1$
$\gamma = 1$
25%
25%
25%
25%
Policy
$V'(s) \leftarrow \sum_a \pi(a|s) \sum_{s'} \sum_r p(s',r|s,a)[r + \gamma V(s')]$
$0.25 * (-1+0) + 0.25 * (-1+0) + 0.25 * (-1+0) + 0.25 * (-1+0) = -1$
V
V'
$v_1 \rightarrow v_2 \rightarrow \dots \rightarrow v_\pi$

(a) (b)

Figure 52: (a) Example of Grid world policy evaluation, (b) Updating step for one grid.

already greedy with respect to its own value function. This is just another way of saying that v_π obeys the Bellman's optimally equation. In which case, π is already optimal.

$$\pi_*(s) = \underset{a}{\operatorname{argmax}} \sum_{s'} \sum_{r} p(s', r|s, a) \left[r + \gamma v_*(s')\right]$$

argmax_a represents the Greedy action.

$$? = \underset{a}{\operatorname{argmax}} \sum_{s'} \sum_{r} p(s', r|s, a) \left[r + \gamma v_\pi(s')\right]$$

$$\pi(s) = \underset{a}{\operatorname{argmax}} \sum_{s'} \sum_{r} p(s', r|s, a) \left[r + \gamma v_\pi(s')\right] \forall s \in S$$

Recall the definition of q_π. It tells you the value of a state if you take action A, and then follow policy π. Imagine we take action A according to π', and then follow policy π. If this action has higher value than the action under π, then π' must be better. The policy improvement theorem formalizes this idea. Policy π' is at least as good as π if in each state, the value of the action selected by π' is greater than or equal to the value of the action selected by π. Policy π' is strictly better if the value is strictly greater and at least one state.

The new policy is a strict improvement over π unless π is already optimal

The policy improvement theorem is as follows.

$$q_\pi(s, \pi'(s)) \geq q_\pi(s, \pi(s)) \forall s \in S \rightarrow \pi' \geq \pi$$

$$q_\pi(s, \pi'(s)) \geq q_\pi(s, \pi(s)) \text{For at least one } s \in S \rightarrow \pi' \geq \pi$$

The final result of the exercise show before is shown in Fig. 53

Policy Iteration

We will show how we can use this to find the optimal policy by iteratively evaluating and proving a sequence of policies.

The policy improvement theorem is.

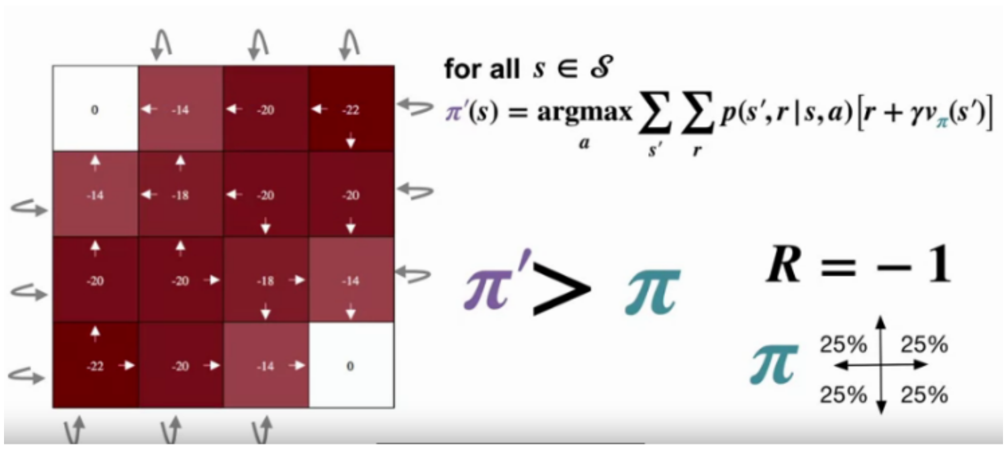

for all $s \in \mathcal{S}$
$\pi'(s) = \underset{a}{\operatorname{argmax}} \sum_{s'} \sum_{r} p(s', r | s, a)[r + \gamma v_\pi(s')]$
$\pi' > \pi$
$R = -1$
π
25%
25%
25%
25%
0
-14
-20
-22
-14
-18
-20
-20
-20
-20
-18
-14
-22
-20
-14
0

Figure 53: Policy evaluation with one array, final result

$$\pi'(s) \doteq \underset{a}{\operatorname{argmax}} \sum_{s'} \sum_{r} p(s', r|s, a) \left[r + \gamma v_\pi(s')\right]$$

π' is strictly better than π unless π was already optimal.

Each policy is guaranteed to be an improvement on the last unless the last policy was already optimal. So when we complete an iteration, and the policy remains unchanged, we know we have found the optimal policy. At that point, we can terminate the algorithm. Each policy generated in this way is deterministic. There are finite number of deterministic policies, so this iterative improvement must eventually reach an optimal policy. This method of finding an optimal policy is called **policy iteration**.

The **policy iteration sequence** is shown in Fig. 54.

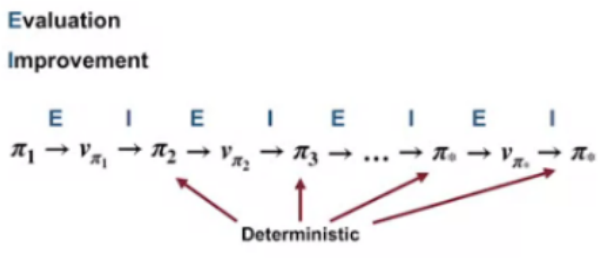

Evaluation
Improvement
E I E I E I E I
$\pi_1 \rightarrow v_{\pi_1} \rightarrow \pi_2 \rightarrow v_{\pi_2} \rightarrow \pi_3 \rightarrow \dots \rightarrow \pi_* \rightarrow v_{\pi_*} \rightarrow \pi_*$
Deterministic

Figure 54: Policy iteration

Policy iteration consists of **two distinct steps** repeated over and over, **evaluation** and **improvement**. We first evaluate our current policy, π_1, which gives us a new value function V that accurately reflects the value of π_1. The improvement step then uses $V_{\pi 1}$ to produce a greedy policy π_2. At this point, π_2 is greedy with respect to the value function of π_1, but V_{π_1} no longer accurately reflects the value of π_2. The next step evaluation makes our value function accurate with respect to the policy π_2.

There is some sort of dance between the value function and the policy π. This dance Fig. 56 leads to an optimal π_* and V_*.

A Pseudo-code is shown in Fig. 57. We initialize V and π in any way we like for each state s. Next, we call iterative policy evaluation to make V reflect the value of π. This is the algorithm we learned earlier in this module. Then, in each state, we set π to select the maximizing action under the value function. If this procedure changes the selected action in any state, we note that the policy is still changing, and set policy

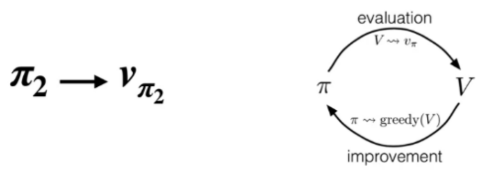

$\pi_2 \longrightarrow v_{\pi_2}$
evaluation
$V \rightsquigarrow v_\pi$
π
V
$\pi \rightsquigarrow \text{greedy}(V)$
improvement

Figure 55: Policy iteration from π_1 to π_2

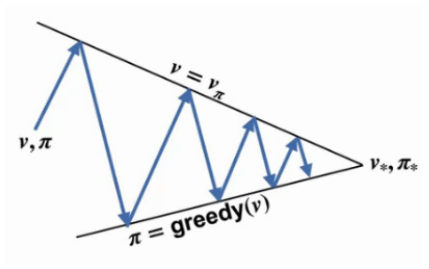
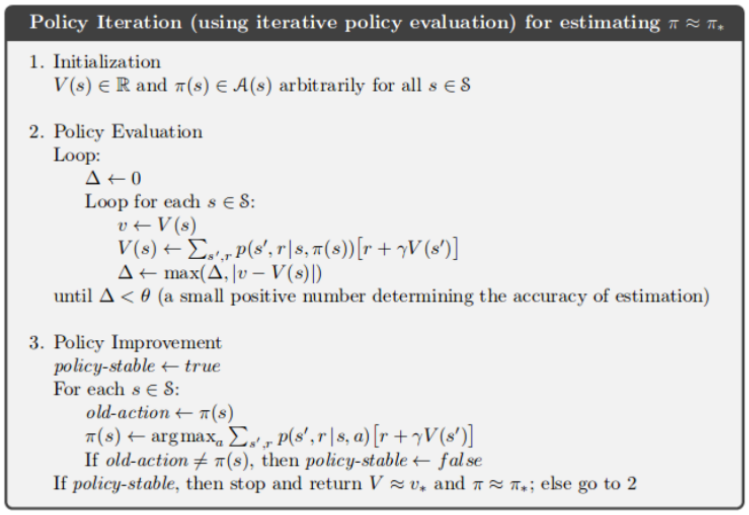


Figure 56: Dance of policy and value

stable to force. After completing step 3, we check if the policy is stable. If not, we carry on and evaluate the new policy.



```
Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Loop:
       $\Delta \leftarrow 0$
       Loop for each $s \in \mathcal{S}$:
           $v \leftarrow V(s)$
           $V(s) \leftarrow \sum_{s',r} p(s', r \mid s, \pi(s))\big[r + \gamma V(s')\big]$
           $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)

3. Policy Improvement
   $policy\text{-}stable \leftarrow true$
   For each $s \in \mathcal{S}$:
       $old\text{-}action \leftarrow \pi(s)$
       $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma V(s')\big]$
       If $old\text{-}action \neq \pi(s)$, then $policy\text{-}stable \leftarrow false$
   If $policy\text{-}stable$, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2
```

Figure 57: Pseudo-code for policy iteration

This example Fig 58 shows the power of policy iteration, in that it guarantees we can follow a sequence of increasingly better policies until we reach an optimal policy. Policy iteration cuts through the search space, which is key when the optimal policy is not straightforward, in this case literally. The same complexity will come up and problems we really care about.

Flexibility of the Policy Iteration Framework

The policy iteration algorithm runs each step all the way to completion. Intuitively, we can imagine relaxing this. Imagine instead, we follow a trajectory like this Fig. 59. Each evaluation step brings our estimate a little closer to the value of the current policy but not all the way. Each policy improvement step makes our policy a little more greedy, but not totally greedy. Intuitively, this process should still make progress

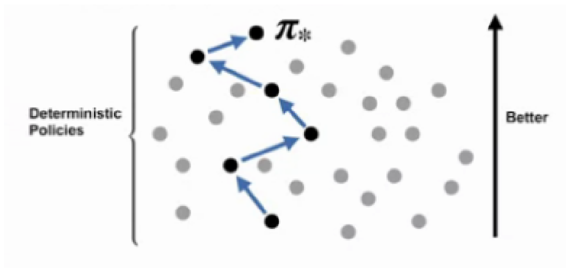


Figure 58: The power of policy iteration

towards the optimal policy and value function. In fact, the theory tells us the same thing. We will use the term generalized policy iteration to refer to all the ways we can interleave policy evaluation and policy improvement.

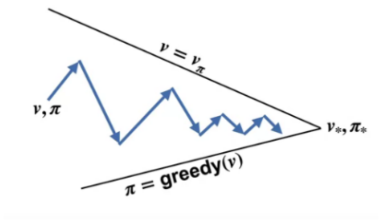
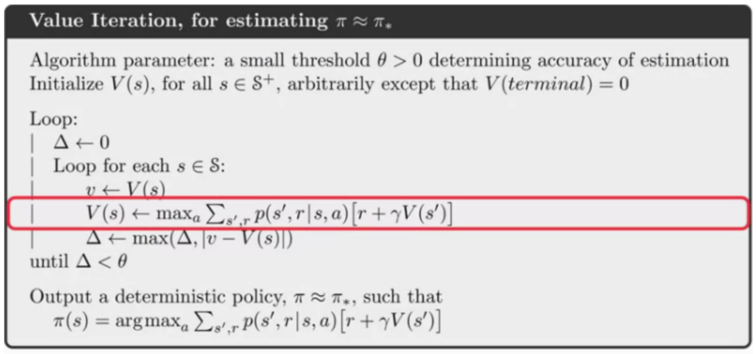


Figure 59: Generalized policy iteration

This brings us to our first generalized policy iteration algorithm, **called value iteration**. In value iteration, we still sweep over all the states and greedify with respect to the current value function. However, we do not run policy evaluation to completion. **We perform just one sweep over all the states**. After that, we greedify again.

We can write this as an update rule Fig 60 which applies directly to the state value function. The update does not reference any specific policy, hence the name **value iteration**. The full algorithm **looks very similar to iterative policy evaluation**. Instead of updating the value according to a fixed falsey, we update using the action that maximizes the current value estimate. Value iteration still converges to V_* in the limit.



Value Iteration, for estimating $\pi \approx \pi_*$

```
Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop:
    $\Delta \leftarrow 0$
    Loop for each $s \in \mathcal{S}$:
        $v \leftarrow V(s)$
        $V(s) \leftarrow \max_a \sum_{s',r} p(s', r | s, a)[r + \gamma V(s')]$
        $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Output a deterministic policy, $\pi \approx \pi_*$, such that
    $\pi(s) = \text{argmax}_a \sum_{s',r} p(s', r | s, a)[r + \gamma V(s')]$
```

Figure 60: Value iteration

Methods that perform systematic sweeps like this are called **synchronous**. This can be problematic if the state space is large. Every sweep could take a very long time. **Asynchronous dynamic programming algorithms update the values of states in any order**, they do not perform systematic sweeps. They might update a given state many times before another is updated even once. In order to guarantee convergence, asynchronous algorithms must continue to update the values of all states. Here for example Fig 61, the algorithm updates the same three states forever ignoring all the others. This is not acceptable because the other states cannot be correct if they are never updated at all. Asynchronous algorithms can propagate value information quickly through selective updates. Sometimes this can be more efficient than a systematic sweep. For example, an asynchronous method can update the states near those that have recently changed value.

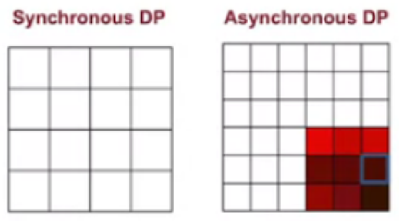


Figure 61: Synchronous and Asynchronous DP

Efficiency of Dynamic Programming

First, recall that the value is the expected return from a given state. The procedure is simple, first, we gather a large number of returns under π and take their average. This will eventually converge to the state value, this is called the Monte Carlo method Fig. 62.

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

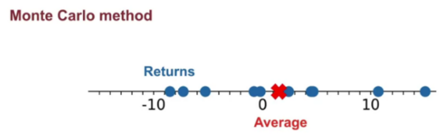


Figure 62: Monte Carlo Method

However, if we do it this way, we may need a large number of returns from each state. Each return depends on many random actions, selected by π, as well as many random state transitions due to the dynamics of the MDP. We could be dealing with a lot of randomness here, each return might be very different than the true state value. So we may need to average many returns before the estimate converges, and we have to do this for every single state.

The key insight of **dynamic programming** is that we do not have to treat the evaluation of each state as a **separate problem**. We can use the other value estimates we have already worked so hard to compute. This process of using the value estimates of successor states to improve our current value estimate is known as **bootstrapping**. **This can be much more efficient than a Monte Carlo** method that estimates each value independently Fig. 63.

$$v_{k+1}(s) \leftarrow \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a) \left[r + \gamma v_k(s')\right]$$

This method simply evaluates every possible deterministic policy one at a time, we then pick the one with the highest value. There are a finite number of deterministic policies, and there always exists an optimal deterministic policy. So brute-force Fig 64 search will find the answer eventually, however, the number of deterministic policies can be huge. A deterministic policy consists of one action choice per state. So the total

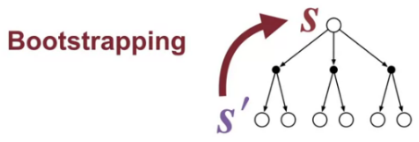


Figure 63: Bootstrapping in dynamic programming

number of **deterministic policies is exponential in the number of states**. Even on a fairly simple problem, this number could be massive, this process could take a very **long time**.

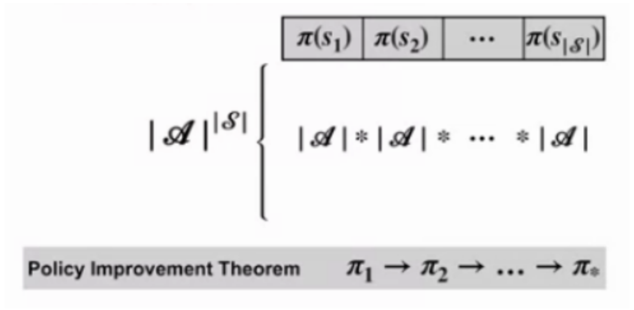

$\pi(s_1)$
$\pi(s_2)$
$\cdots$
$\pi(s_{|\mathcal{S}|})$
$|\mathcal{A}|^{|\mathcal{S}|}$
$|\mathcal{A}| * |\mathcal{A}| * \cdots * |\mathcal{A}|$
Policy Improvement Theorem
$\pi_1 \rightarrow \pi_2 \rightarrow \ldots \rightarrow \pi_*$

Figure 64: Brute-force Search

There is a problem with the computation speed when using brute-force search. The efficiency of policy iteration and brute-force search is compared in Fig 65

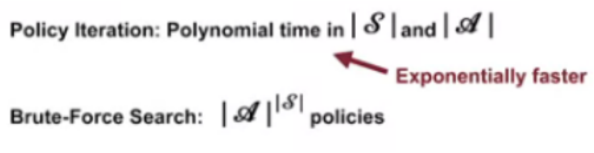

Policy Iteration: Polynomial time in $|\mathcal{S}|$ and $|\mathcal{A}|$
Exponentially faster
Brute-Force Search: $|\mathcal{A}|^{|\mathcal{S}|}$ policies

Figure 65: Policy iteration vs Brute-force search

The curse of dimensionality

- The size of the state space grows exponentially as the number of relevant features increases
- This is not an issue with Dynamic Programming, but an inherent complexity of the problem

References

[1] University of Alberta. *Fundamentals of Reinforcement Learning*. 2019. URL: `https://www.coursera.org/learn/fundamentals-of-reinforcement-learning` (visited on 11/12/2019).

[2] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018.