Introduction to Dynamic Programming

- What is Dynamic programming?
- Policy evaluation
- State-value policy evaluation
- Action-value policy evaluation
- Control or Policy improvement
- Policy iteration
- Value iteration

Introduction to Dynamic Programming

- **Dynamic programming** was developed by the mathematician Richard Bellman in the early 1950s
- Dynamic programming is a **optimal planning method**
  - Planning methods use a **model of the environment**
  - Environment model is **Markov process**
  - Achieve a **goal** given a **model of the environment**
- Planning methods enable an intelligent agent to **gain autonomy**
  - Perform a sequence of **optimal actions**
  - Follow **plan** or **policy**

Introduction to Dynamic Programming

- Why did Bellman give this method its name?
  - **Programming** is a computer algorithm which creates a **plan of actions** to optimize the **utility** or **total reward**
  - A **Dynamic** algorithm solves the problem recursively, operating on smaller and simpler sub-problems
- DP methods are scalable, practical and widely used
  - Schedule optimization
  - Optimal roughting
  - Optimal control
  - Etc.

Introduction to Dynamic Programming

- Like DP, **reinforcement learning** is a class of optimization algorithms to optimize utility in a system represented by a Markov processes
- For many problems intelligent agents can use either DP or RL
  - DP requires **model specification**
  - RL is **model free**
  - Both DP and RL use **bootstrapping algorithms**

Policy Evaluation

- We want our intelligent agent to follow an **optimal policy**
- **Policy evaluation** is needed to compare policy
- Can evaluate policy by value:
  - **State value**: expected value of being in a state
  - **Action value**: expected value of taking an action in a given state

Policy Evaluation

- Recall the definition of **discounted return** from the current time t

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

- Where,

  R_{t+1} = the expected reward from state S_t to state $S_{t+1} = E\left[R_{t+1} \mid S_t = s\right]$

  γ = discount factor

State Value Policy Evaluation

- **Bellman value equations** are fundamental to computing expected state values
- We can find the state value, of state s, given a policy π as the expected value of the gain:

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

Where $\mathbb{E}_\pi[] =$ Expectation given policy π

State Value Policy Evaluation

Expand the Bellman value equations to find a **recursion**

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

Since **gain equals the reward plus the gain at the next step**:

$$v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s]$$

Finally, **bootstrap by approximating gain by state value**, at the next step:

$$v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s]$$

State Value Policy Evaluation

Compute the expected value for the Bellman value equations

$$
\begin{aligned}
v_\pi(s) &= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s] \\
&= \sum_a \pi(a|s) \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma v_\pi(s')\right], \ \forall a
\end{aligned}
$$

Where

$a =$ an action by the agent

$\pi(a|s) =$ the policy specifying action, a, given state, s

$p(s', r \mid s, a) =$ probability of successor state, s', and reward, r, given state, s, and action a

$\left[r + \gamma v_\pi(s')\right] =$ the bootstrapped state value

State Value Policy Evaluation

- There is one Bellman value equation for each state, s or n equations
- In theory this system of equations can be solved directly

  But requires O(n^3) computations
- Or, can use the recursion relationship using the last estimate of $v_\pi(s')$

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

- Using the estimated state value to compute a better estimate is called **bootstrapping**

Action Value Policy Evaluation

- **Bellman action value equations** are fundamental to computing expected action values
- We can find the action value, of taking action a in state s, given a policy π as

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

Where $\mathbb{E}_\pi[] = \text{Expectation given policy } \pi$

Action Value Policy Evaluation

Expand the Bellman action value equations to find a **recursion**

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

Expanding the **gain as the sum of rewards** gives:

$$q_\pi(s,a) = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k \, R_{t+k+1} \,\middle|\, S_t = s, A_t = a\right]$$

Then using the transition probability and the **bootstrapped state values** gives:

$$q_\pi(s,a) = \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma \, q_\pi(S_{t+1}, a')\right]$$

Where, $A_t =$ the action taken at step t

$a' =$ the action taken from the successor state, s'

Control: Policy Improvement

- Need a way to **improve value of a policy**
- Ideally want an **optimal policy**
- **Policy improvement theorem** says a optimal policy has **the highest value of any possible policy**
- For **state values** the **policy improvement theorem** is:

$$v_*(s) >= v_\pi(s) \ \forall \pi$$

Where $v_*(s)$ is the optimal policy

Control: Policy Improvement

- Need a way to **improve value of a policy**
- Ideally want an **optimal policy**
- **Policy improvement theorem** says a optimal policy has the highest value of any possible policy.
- For **action values** the policy improvement theorem becomes:

$$q_*(s,\ a) >= q_\pi(s,\ a)\ \forall\ \pi$$

Where $q_*(s,\ a)$ is the optimal action value policy

Policy Iteration

- Need to find an algorithm to compute a policy π_* such that:

$$v_*(s) >= v_\pi(s) \; \forall \pi$$

- The **Bellman state value equations** are a **bootstrap** formulation for computing optimal policy

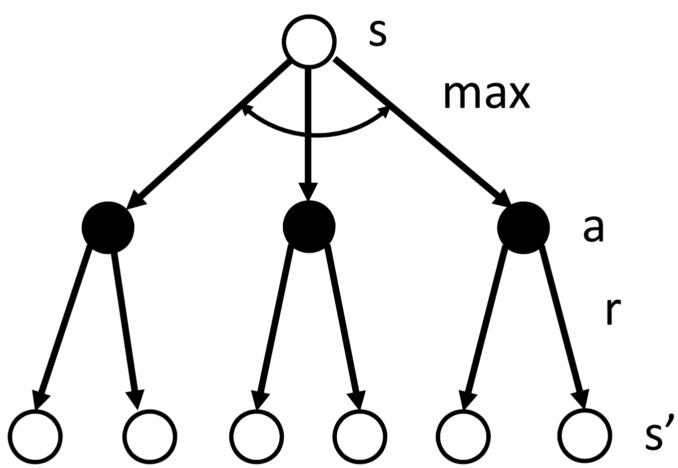
$$\begin{aligned} v_*(s) &= max_\pi v_\pi(s) \\ &= max_a \; \mathbb{E}[G_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] \\ &= max_a \; \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma v_*(s')\right] \end{aligned}$$

Policy Iteration

- How to understand the **bootstrap Bellman state value equations**?

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

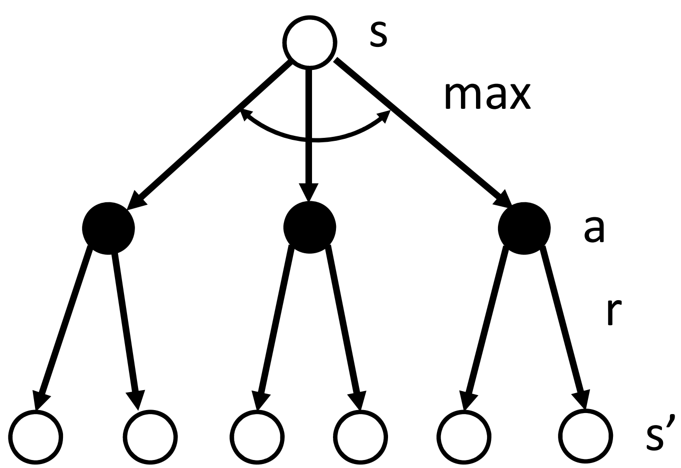
- Use a **backup diagram**:



1. Start Markov process in state s:
   state = Open circle
2. Take action a that maximizes state value:
   action = Filled circle
3. Leads to successor states s’ with reward r

Policy Iteration

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



- Policy iteration **backs up** into a better estimates of state value by looking one step ahead
- Since the reward for all actions must be computed, the algorithm is said to use a **full backup**
- The computations for the full backup grow with the number of actions and successor states
- Bellman called this property the **curse of dimensionality**

Value Iteration

- Need to find an algorithm to compute a policy π_* such that:

$$q_*(s,\ a) >= q_\pi(s,\ a)\ \forall\ \pi$$

- The Bellman **action value equations** are an **bootstrap** formulation for computing optimal policy

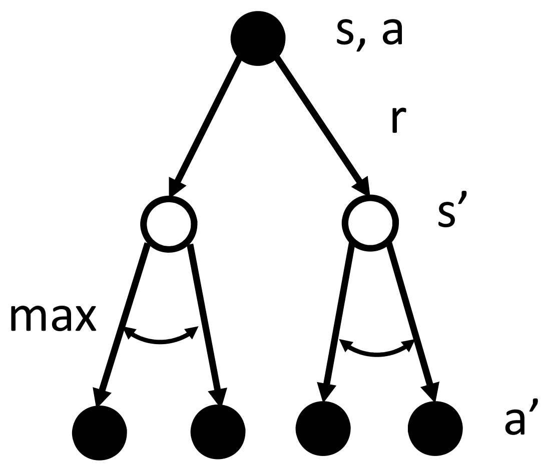
$$\begin{aligned} q_*(s,a) &= max_\pi q(s,a) \\ &= \mathbb{E}\Big[R_{t+1} + \gamma max_{a'}\ q_*(S_{t+1}, a')\ \Big|\ S_t = s, A_t = a\Big] \\ &= max_a \sum_{s',r} p(s', r \mid s, a)\Big[r + \gamma\ max_{a'}\ q_*(S_{t+1}, a')\Big] \end{aligned}$$

Value Iteration

- How to understand the Bellman **bootstrap action value equations**?

$$q_*(s,a) = max_a \sum_{s',r} p(s',r \mid s,a)\Big[r + \gamma\, max_{a'}\; q_*(S_{t+1},a')\Big]$$

- Use a **backup diagram** to understand this method:



1. Start Markov process with state action tuple (s,a)

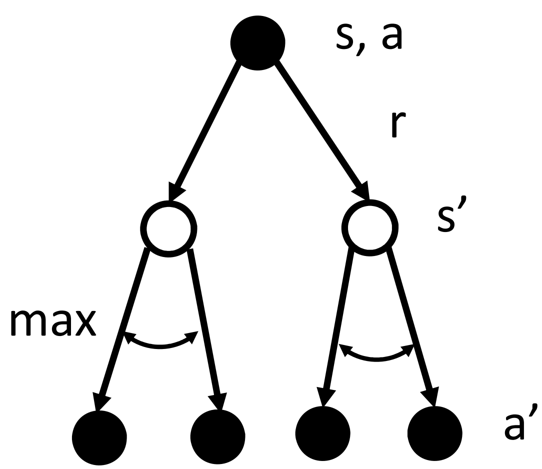
2. Leads to successor states, s’, with reward r

3. Successor action, a’, that maximizes expected value

4. Maximum of successor action value is used to bootstrap next optimal action value

Value Iteration

$$q_*(s,a)\, max_a \sum_{s',r} p(s',r \mid s,a)\Big[r + \gamma\, max_{a'}\; q_*(S_{t+1}, a')\Big]$$



- Value iteration **backs up** into a better estimates of action value by looking one step ahead
- Since the reward for all successor state action pairs is computed, the algorithm is said to use a **full backup**
- The computations for the full backup grow with the number of actions and successor states
- Same **curse of dimensionality** as policy iteration