Introduction to Deep Reinforcement Learning

Model-free Methods

Zhiao Huang

Deep Reinforcement Learning Era

- In 2013, DeepMind uses Deep Reinforcement learning to play Atari Games

DeepMind

ATARI 2600

Mnih, Volodymyr, et al. "**Playing atari with deep reinforcement learning**." *arXiv preprint arXiv:1312.5602* (2013)

Deep Reinforcement Learning Era

- In March 2016, Alpha Go beat the human champion Lee Sedol

Silver, David, et al. "**Mastering the game of go without human knowledge**." *Nature* 550.7676 (2017): 354-359.

Deep RL for Robotics

- People started to apply deep learning into robotics

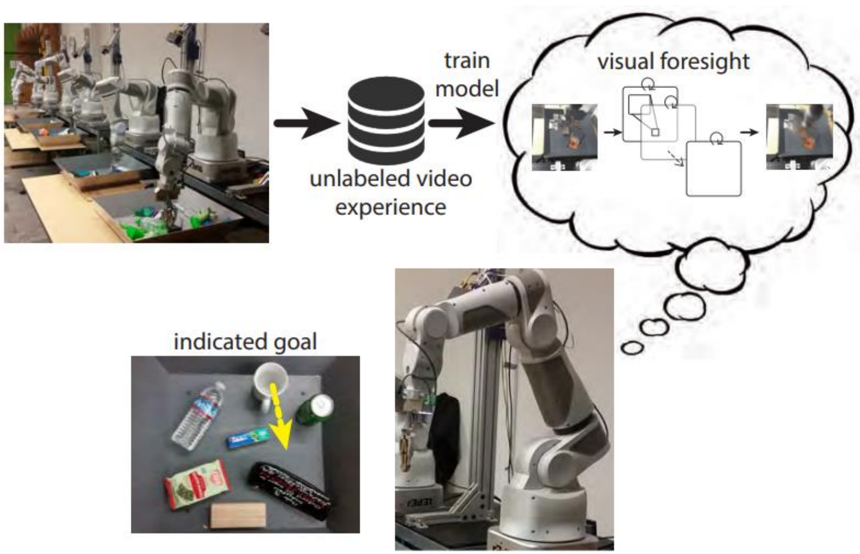


Fig. 1. Using our approach, a robot uses a learned predictive model of images, i.e. a visual imagination, to push objects to desired locations.

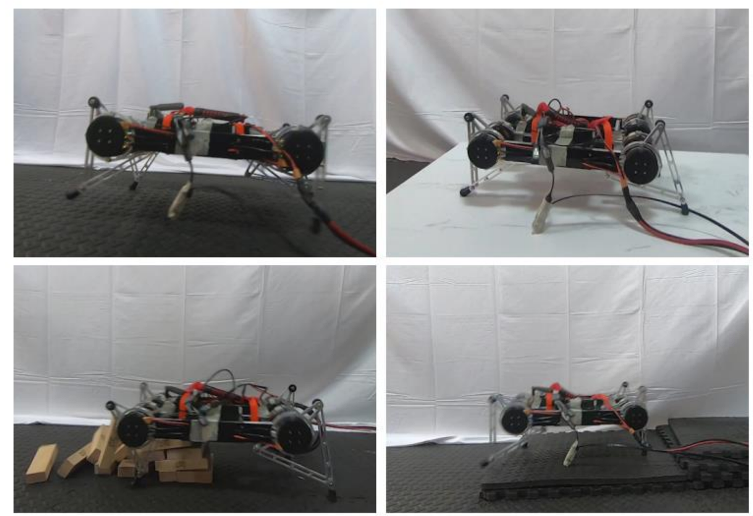


Fig. 1: Illustration of a walking gait learned in the real world. The policy is trained only on a flat terrain, but the learned gait is robust and can handle obstacles that were not seen during training.

Levine, Sergey, and Chelsea Finn. "**Deep visual foresight for planning robot motion**." (2017).

Haarnoja, Tuomas, et al. "**Learning to walk via deep reinforcement learning**." *arXiv preprint arXiv:1812.11103* (2018).

So…

What is deep reinforcement learning?

Why do we care about it?

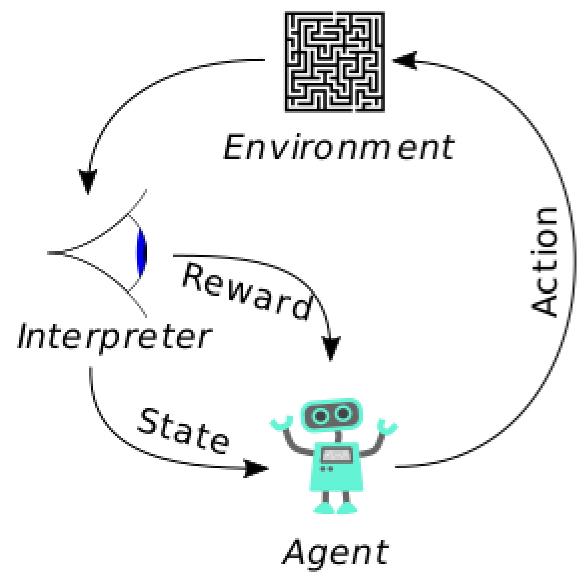
Overview

- Introduction
  - Markov Decision Process
  - Policy, value and the Bellman-Ford equation
- Q-learning
  - DQN/DDPG
- Policy Optimization
  - REINFORCE
  - Actor-Critic Methods

Reinforcement Learning

Machine Learning

- Supervised Learning
- Unsupervised Learning
- Reinforcement Learning



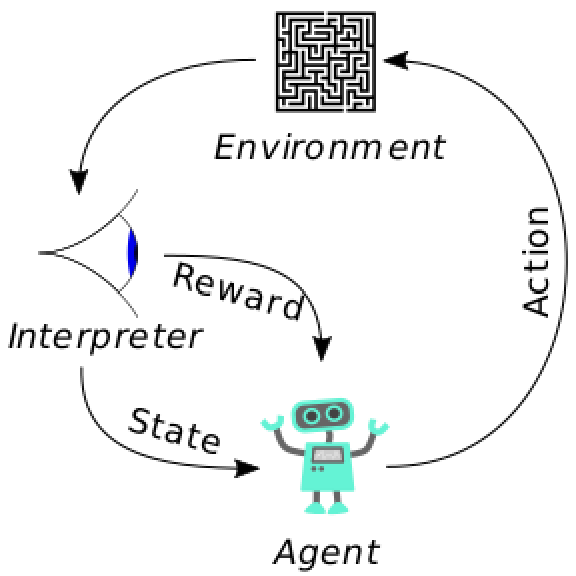
Reinforcement learning is modeled as a

Markov Decision Process

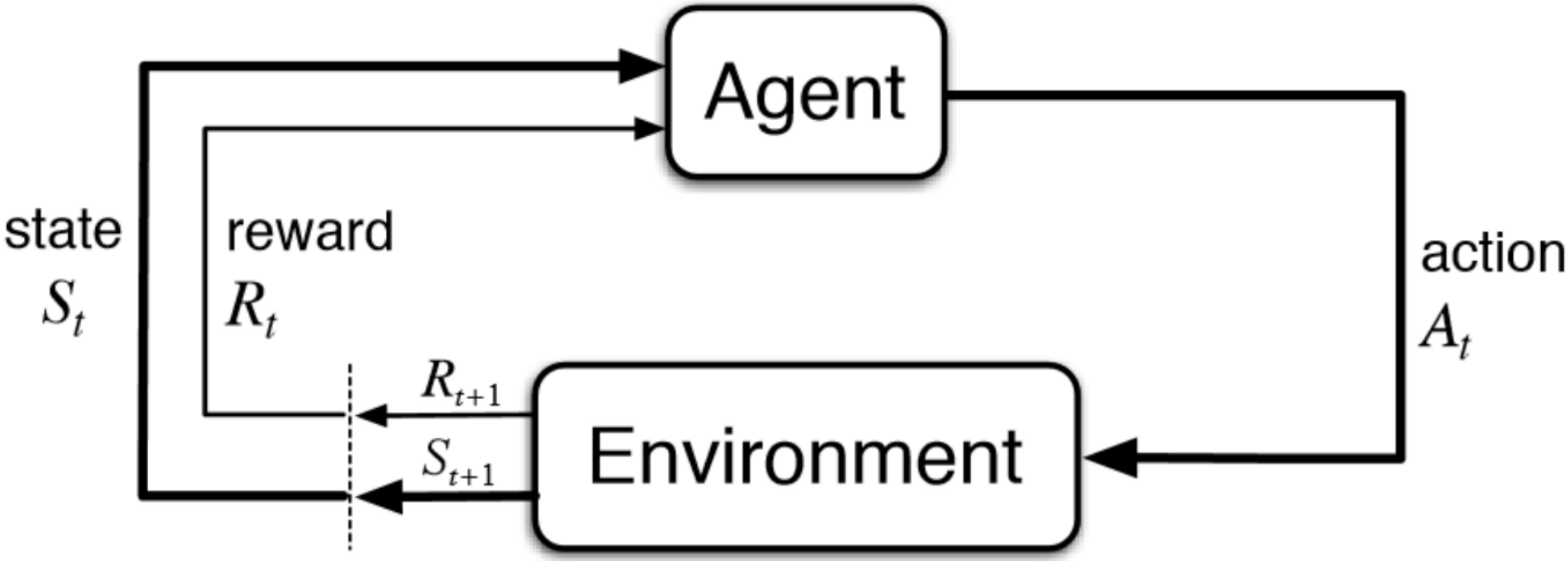
Markov Decision Process

Reinforcement learning (RL) is an area of machine learning concerned with how software agents ought to take actions in an environment in order to maximize the notion of cumulative reward.

Basic reinforcement learning is modeled as a **Markov Decision Process**



Markov Decision Process



Markov Decision Process

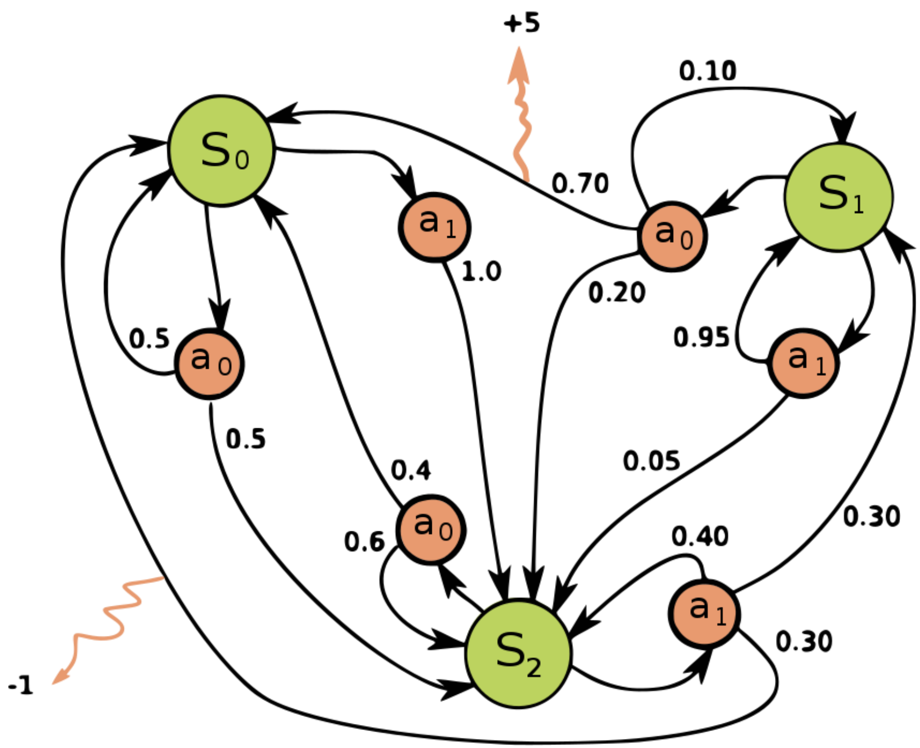
Reinforcement Learning is modeled as a Markov Decision Process S, A, P_a, R_a

- A set of environment and agent state, S;
- A set of actions, A, of the agent;
- $P_a(s, s') = \Pr(s_{t+1} = s' | s_t = s, a_t = a)$ is the probability of transition (at time t) from state s to state s' under action a (**Markov property**)
- $R_a(s, s') = R(s, a, s')$ is the immediate reward received after transitioning from state s to s', under action a.

Note that the transition probability is identical in each time step…

An Example in Graph

In discrete and finite cases, MDP≈Graph

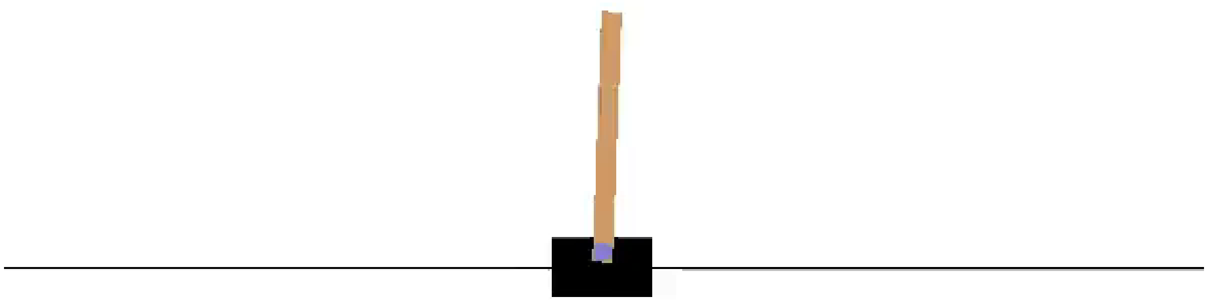


An Example in Python

OpenAI Gym Interface

```python
import gym
env = gym.make('CartPole-v0')
observation = env.reset()
for t in range(100):
    action = env.action_space.sample()
    observation, reward, done, info = env.step(action)
    if done:
        break
env.close()
```

An Example in Python



Finite-horizon vs. Infinite horizon

- MDP itself doesn’t define a time limit …

```
import gym
env = gym.make('CartPole-v0')
observation = env.reset()
for t in range(100):
    action = env.action_space.sample()
    observation, reward, done, info = env.step(action)
    if done:
        break
env.close()
```

Finite-horizon vs. Infinite horizon

- MDP itself doesn't define a time limit … which is called an infinite horizon process

- But we can turn the finite horizon environment into an infinite horizon environment by adding an extra terminal state and the current time step into the observation …

- In practice, we usually set a maximum time step:

```
for t in range(100):
```

Overview

- Introduction
  - Markov Decision Process
  - Policy, value and the Bellman-Ford equation
- Q-learning
  - DQN/DDPG
- Policy Optimization
  - REINFORCE
  - Actor-Critic Methods

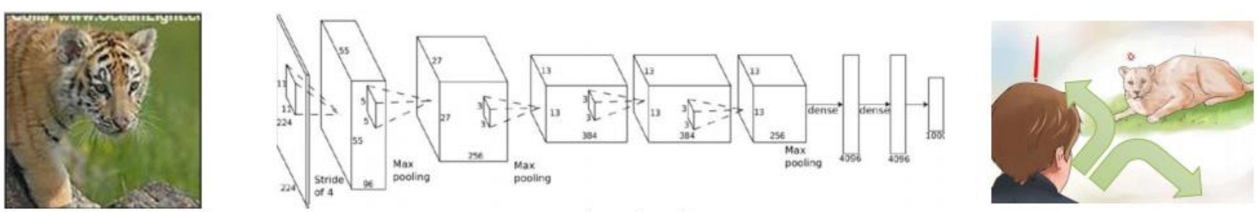
The solution to the MDP is a **policy** that maximize the expected total reward

Policy

- Policy is a global mapping from states to action:

$$\pi(s): S \rightarrow A$$

- Policy can be a neural network:



$\mathbf{o}_t$ $\pi_\theta(\mathbf{a}_t|\mathbf{o}_t)$ $\mathbf{a}_t$

$\mathbf{s}_t$ – state
$\mathbf{o}_t$ – observation
$\mathbf{a}_t$ – action

$\pi_\theta(\mathbf{a}_t|\mathbf{o}_t)$ – policy
$\pi_\theta(\mathbf{a}_t|\mathbf{s}_t)$ – policy (fully observed)

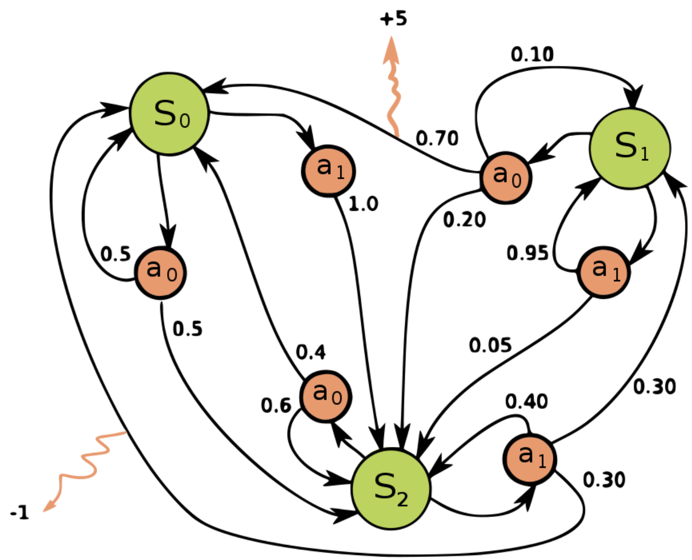
Policy

- Policy is a global mapping from states to actions:

$$\pi(s): S \to A$$

- Sometimes we consider a random policy,

$$\pi(a|s): S \times A \to \mathbb{R}$$



$P(a\|s)$	a_0	a_1
s_0	0.5	0.5
s_1	0.0	1.0
s_2	1.0	0.0

Policy in Python

- Policy is a global mapping from states to actions:

$$\pi(s): S \rightarrow A$$

```
env = gym.make('CartPole-v0')
observation = env.reset()

policy = lambda x: 1 if random.random() < x[0] else 0

for t in range(100):
    action = policy(observation)
    observation, reward, done, info = env.step(action)
    if done:
        break
```

Trajectory

- Every time we "sample" the action by the policy in the environment, we get a **trajectory**:

$$\tau = \{s_0, a_0, s_1, a_1, \dots, s_{t-1}, a_{t-1}, \dots\}$$

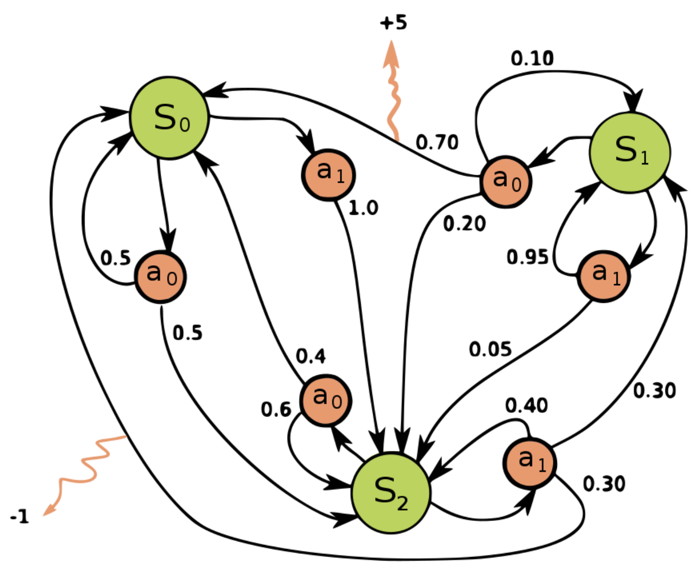
```python
env = gym.make('CartPole-v0')
observation = env.reset()

policy = lambda x: 1 if random.random() < x[0] else 0

for t in range(100):
    action = policy(observation)
    observation, reward, done, info = env.step(action)
    if done:
        break
```

Trajectory

- Every time we "sample" the action by the policy in the environment, we get a **trajectory**:

$$\tau = \{s_0, a_0, s_1, a_1, \dots, s_{t-1}, a_{t-1}, \dots\}$$

For example:



A trajectory:

$$S_0, a_0, S_1, a_1, S_2, \dots$$

Trajectory

- Both the policy and the MDP are random!
- If we know $\pi(a_i|s_i)$ and $p(s_{i+1}|s_i, a_i)$, we can compute the probability of a sampled trajectory:

$$p(\tau) = p(s_0) \prod_{i=0\ldots\infty} \pi(a_i|s_i) p(s_{i+1}|s_i, a_i)$$

- We don't like products, so we usually consider the log-likelihood

$$\log p(\tau) = \log p(s_0) + \sum_{i=0\ldots\infty} \log \pi(a_i|s_i) + \log p(s_{i+1}|s_i, a_i)$$

Trajectory

- A policy will induce a distribution over the trajectories

$\tau \sim p_\pi(\tau)$ or $\tau \sim p_\pi(\tau|s_0)$

- This notation will be useful later

The solution to the MDP is a policy that maximizes the expected total **reward**

Reward

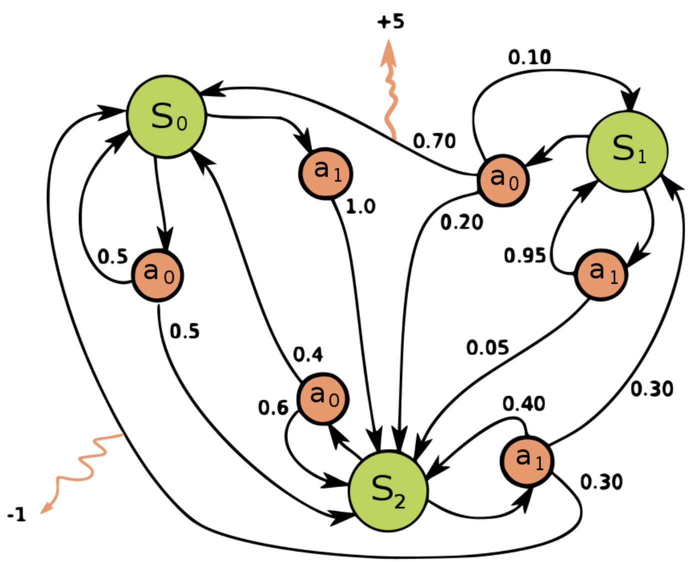
- The goal of the reinforcement learning is to find a policy to maximize the **total reward**

```
total_reward = 0
for t in range(100):
    action = policy(observation)
    observation, reward, done, info = env.step(action)
    total_reward += reward
    if done:
        break
print(total_reward)
```

Reward

- The goal of the reinforcement learning is to find a policy to maximize the **total reward**

- In the discrete case, the total reward is similar to the length of the path



Reward

- The total reward of a trajectory is the sum of the rewards on each time step

$$R(\tau) = \sum_{i=0\ldots\infty} R(s_i, a_i, s_{i+1})$$

- If we fix the policy and the starting state, we can calculate the expected rewards of the sampled trajectories:

$$V_\pi(s_0) = E_{\tau \sim p_\pi(\tau | s_0)}[R(\tau)]$$

Discount Factor

- The total reward of a trajectory is the sum of the rewards at each time step

$$R(\tau) = \sum_{i=0\ldots\infty} R(s_i, a_i, s_{i+1})$$

- It may not converge for infinite-horizon trajectories

Discount Factor

- Let $0 \leq \gamma < 1$ and
$$R_\gamma(\tau) = \sum_{i=0\ldots\infty} \gamma^i R(s_i, a_i, s_{i+1}) \leq \frac{R_{max}}{1-\gamma}$$
- We always add a **discount factor** into the rewards
- γ is usually omitted from the notation

RL Problem

Formally, given an MDP, find policy π to maximize the expected future reward:

$$\max_{\pi} E_{\tau \sim p_{\pi}(\tau)}\left[R_{\gamma}(\tau)\right]$$

where

$$p(\tau) = p(s_0) \prod_{i=0\ldots\infty} \pi(a_i|s_i) p(s_{i+1}|s_i, a_i)\,,$$

$$R_{\gamma}(\tau) = \sum_{i=0\ldots\infty} \gamma^i R(s_{i+1}|s_i, a_i)$$

How can we find the **optimal** one?

Naïve Algorithm 1: Random search

We can random sample π and evaluate it with Monte-Carlo sampling:

$$\max_{\pi} E_{\tau \sim p_{\pi}(\tau)}\left[R_{\gamma}(\tau)\right]$$

- For i from 1 to ∞
  - Sample policy π
  - Sample τ to evaluate $R_{\pi} = E_{\tau \sim p_{\pi}(\tau)}\left[R_{\gamma}(\tau)\right]$
  - Return π if R_{π} is large enough

Naïve Algorithm 1: Random search

We can random sample π and evaluate it with Monte-Carlo sampling:

$$\max_{\pi} E_{\tau\sim p_{\pi}(\tau)}[R_{\gamma}(\tau)]$$

We can speed it up with value iteration

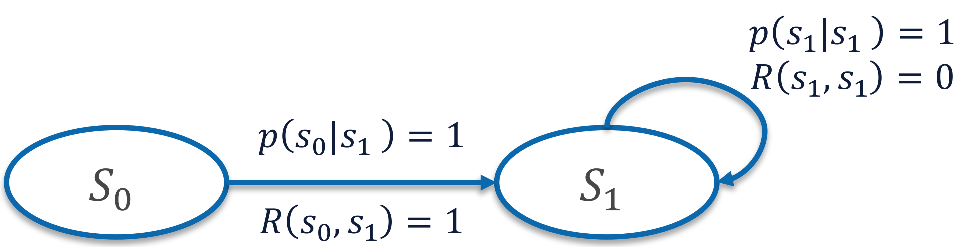
- For i from 1 to ∞
  - Sample policy π
  - Sample τ to evaluate $R_{\pi} = E_{\tau\sim p_{\pi}(\tau)}[R_{\gamma}(\tau)]$
  - Return π if R_{π} is large enough

Value Function

- Fix π, if we marginalize out the action a, for one step transition:

$$p(s'|s) = \sum_a \pi(a|s)\, p(s'|s, a)$$

$$R(s, s') = \sum_a R(s, a, s')\pi(a|s)$$



$$\sum_{s_0, s_1, \ldots} \prod_i p(s_{i+1}|s_i) \sum_i R(s_i, s_{i+1}) = \sum_\tau p(\tau) R(\tau)$$

Value Function

- Value function

$$V_\pi(s) = E_{\tau \sim p_\pi(\tau|s)}[R(\tau)]$$

- The expected reward if the agent follows the policy π and is currently at the state s

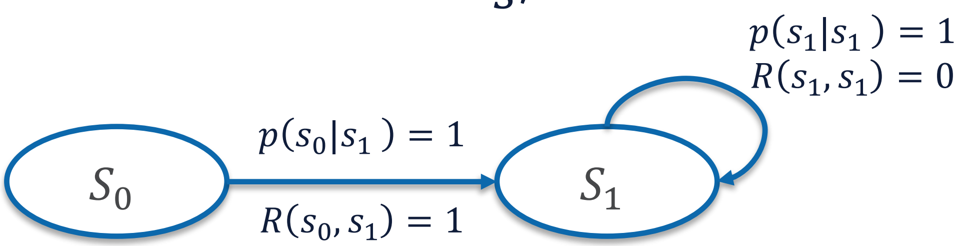
- It's well-defined because of the Markov property

Bellman Equations (1)

- V_π satisfies the **Bellman equations** by definition:

$$V_\pi(s) = E_{a\sim\pi(s),s'\sim p(s'|s,a)}[R(s,a,s') + \gamma V_\pi(s')]$$

$$= \sum_{s'} R(s,s') + \gamma p(s,s')V_\pi(s')$$



$$V_0 = 1 + \gamma V_1$$
$$V_1 = 0 + \gamma V_1$$

Value Iteration

Bellman equation is a linear operator:

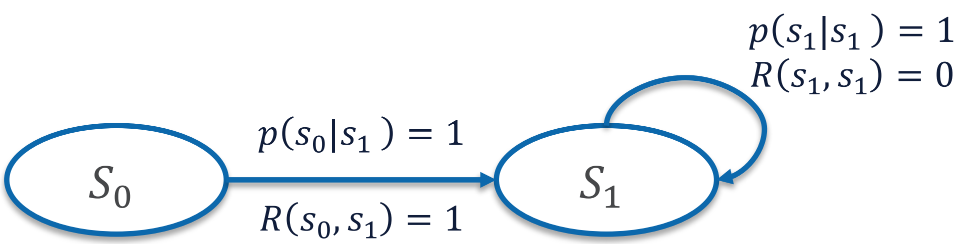
$$V_\pi(s) = \sum_{s'} R(s, s') + \gamma p(s, s') V_\pi(s')$$
$$\Rightarrow V_\pi = R + \gamma P V_\pi$$

Let $f(x) = R + \gamma P x$, then x is the fixed point of f.

> Initialize with V^0
> For I from 1 to ∞:
>   In each step, let $V^{t+1} = f(V^t)$
>   Return V when it converges.

Example

- $\mathrm{R} = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, \mathrm{B} = 0.99 \begin{bmatrix} 0 & 1 \\ 0 & 1 \end{bmatrix}$, we can check for any V^0, the iteration will converge at $V = [1, 0]$.




```
import numpy as np

R = np.array([1, 0])
B = 0.99 * np.array([[0, 1], [0, 1]])

V = np.random.random((2,))
for i in range(100000):
    V = R+B.dot(V)
print(V)
```

- f is a contraction: let V' be the real value function

$$\|f(V) - f(V')\|_\infty \leq \gamma \|PV - PV'\| \leq \gamma \|V - V'\|$$

Naïve Algorithm 2: Policy Evaluation

We can random sample π and evaluate it with value iteration:

$$\max_{\pi} E_{\tau\sim p_{\pi}(\tau)}\left[R_{\gamma}(\tau)\right]$$

- For i from 1 to ∞
  - Sample policy π
  - Value iteration for $R_{\pi} = E_{\tau\sim p_{\pi}(\tau)}\left[R_{\gamma}(\tau)\right]$
  - Return π if R_{π} is large enough

Better Approaches

We can random sample π and evaluate it with value iteration:

$$\max_{\pi} E_{\tau \sim p_{\pi}(\tau)}\left[R_{\gamma}(\tau)\right]$$

- For i from 1 to ∞
  - ~~Sample policy π~~ Optimize policy π
  - Value iteration for $R_{\pi} = E_{\tau \sim p_{\pi}(\tau)}\left[R_{\gamma}(\tau)\right]$
  - Return π if R_{π} is large enough

Q function

- For all state s and action a, we can define Q function

$$Q_\pi(s,a) = E_{s\prime\sim p(s'|s,a)} R(s,a,s') + \gamma V_\pi(s')$$

$$\Rightarrow V_\pi(s) = \underset{a\sim\pi(s)}{E} Q_\pi(s,a)$$

Expected reward from state s if we select the action a.

Bellman Equations (2)

- Consider the optimal policy π^*, we must have

$$\pi^*(s) = \text{argmax}_a Q_{\pi^*}(s,a)$$

Where $Q_{\pi^*}(s,a) = E_{s\prime\sim p(s'|s,a)} R(s,a,s') + \gamma V_{\pi^*}(s')$

- Otherwise let $\pi'(\text{s}) = \text{argmax}_a Q(s,a)$ produces a better policy

$$(R' + \gamma P'V)(s) > (R^* + \gamma P^*V)(s)$$
$$f'(V) = R' + \gamma P'V > R^* + \gamma P^*V = f^*(V)$$

Bellman Equations (2)

Put them together:

$$\pi^*(s) = \text{argmax}_a Q(s,a)$$

$$Q_{\pi^*}(s,a) = E_{s \sim p(s'|s,a)} R(s,a,s') + \gamma V_{\pi^*}(s')$$

$$V_{\pi^*} = E_{a \sim \pi^*(s)}[Q_{\pi^*}(s,a)]$$

$$\Rightarrow V^*(s) = \max_a E_{s' \sim p(s,a)}[R(s,a,s') + \gamma V^*(s')]$$

Bellman Equations (2)

- The **Bellman equations (2)**:

$$V^*(s) = \max_a E_{s' \sim p(s,a)}[R(s,a,s') + \gamma V^*(s')]$$

- The same with the expectation case, we can prove that this is a contraction, so it has a unique fixed point.

Bellman Equations (2)

- The **Bellman equations (2)**:

$$V^*(s) = \max_a E_{s' \sim p(s,a)}[R(s,a,s') + \gamma V^*(s')]$$

- We only need to find the policy that satisfies:

$$\pi(s) = \mathrm{argmax}_a Q_\pi(s,a)$$

which is a fixed point of the bellman equations

Algorithm 3: Policy Iteration

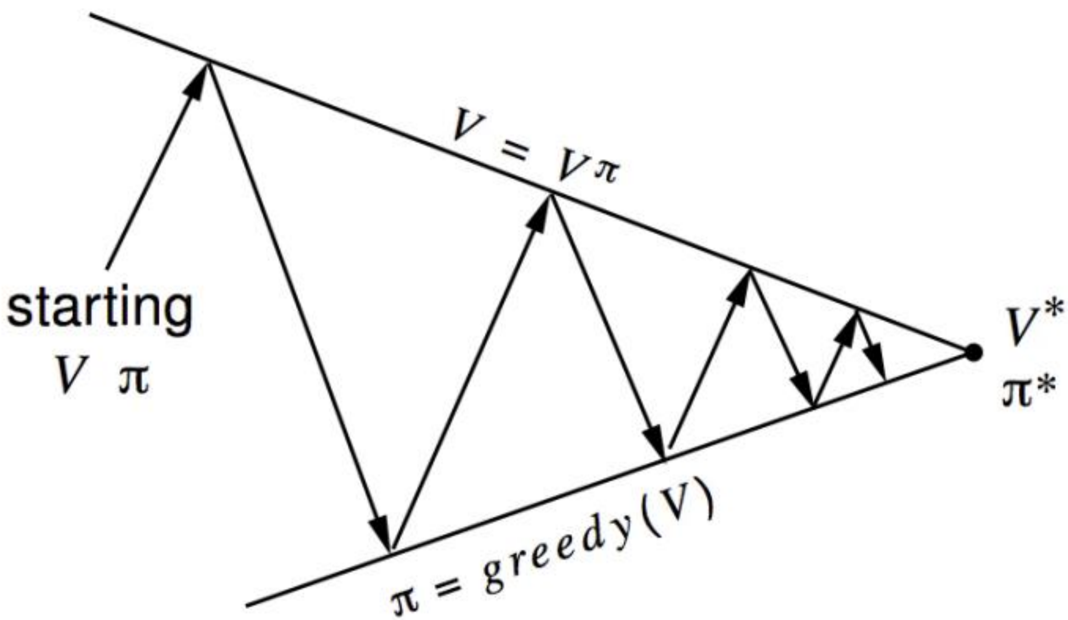
- We only need to find the policy that satisfies:

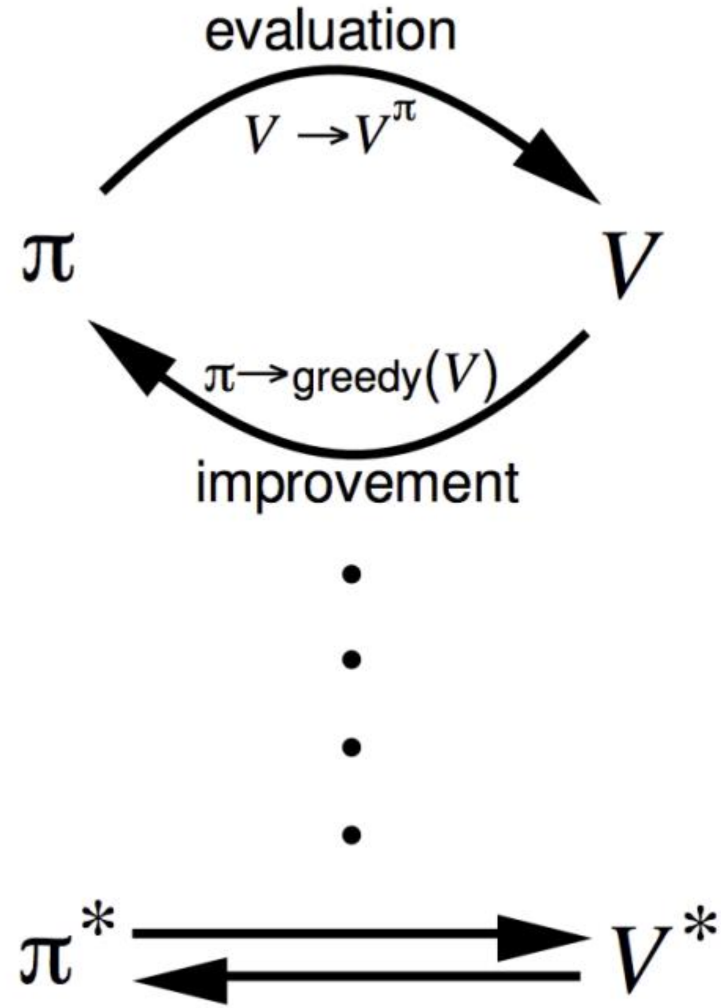
$$\pi(s) = \text{argmax}_a Q_\pi(s, a)$$

- Policy iteration methods iteratively adjust the policy for each state to satisfy the above constraints

- For i from 1 to ∞
  - Compute V_π and Q_π with the value iteration
  - $\forall s, \pi(s) = \text{argmax}_a Q_\pi(s, a)$
  - Return π if it converges

Policy Iteration





Value Iteration

- We can consider the previous algorithm in the view of value function
- The optimal value function $V^* = V_{\pi^*}$ should satisfy the Bellman Equation for all states:

$$V^*(s) = \max_a [E_{s' \sim p(s,a)}[R(s,a,s') + \gamma V^*(s')]]$$

- V^* is the fixed point of the operator $B(V)$
  - Find the fixed point iteratively

Algorithm 4: Value Iteration

repeat

$U \leftarrow U'; \delta \leftarrow 0$

for each state s **in** S **do**

$$U'[s] \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' \mid s, a)\, U[s']$$

if $|U'[s] - U[s]| > \delta$ **then** $\delta \leftarrow |U'[s] - U[s]|$

Value Iteration and the Policy Iteration

- Value iteration and the policy iteration all try to solve the Bellman Equation;

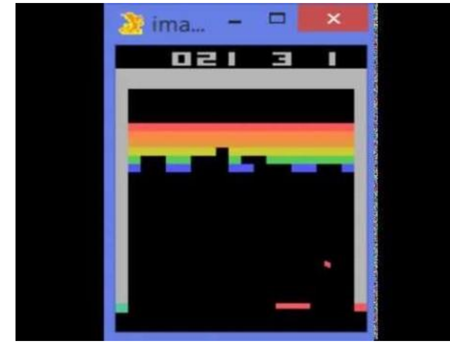
- Policy Iteration methods: maintain the value functions for the current policy, and then adjust the policy based on

$$\pi(s) = \mathrm{argmax}_a Q_\pi(s, a)$$

- Value Iteration methods: solve V^* directly with

$$V^*(s) = \max_a E_{s' \sim p(s,a)}[R(s, a, s') + \gamma V^*(s')]$$

What's the trouble?



- Image input

- S is a high-dimensional space
  - We can't maintain π, Q and V directly

- The model $p(s'|s,a)$ and the reward $R(s,a,s')$ is unknown
  - we can only sample trajectories from the environment with the policy

Next

- Introduction
  - Markov Decision Process
  - Policy, Value and the Bellman-Ford equation
- Deep Q-learning
  - DQN/DDPG
- Policy Optimization
  - REINFORCE
  - Actor-Critic Methods