Machine Learning Department
School of Computer Science
Carnegie Mellon University

Reinforcement Learning:
Value Iteration & Policy Iteration

Matt Gormley & Henry Chai
Lecture 21
Nov. 11, 2024

Reminders

- **Homework 7: Deep Learning**
  - Out: Fri, Nov. 8
  - Due: Sun, Nov. 17 at 11:59pm
- **Homework 8: Deep RL**
  - Out: Sun, Nov. 17
  - Due: Mon, Nov. 25 at 11:59pm

MARKOV DECISION PROCESSES

RL: Components

From the Environment (i.e. the MDP)

- State space, $\mathcal{S}$
- Action space, $\mathcal{A}$
- Reward function, $R(s,a),\ R:\ \mathcal{S} \times \mathcal{A} \to \mathbb{R}$
- Transition probabilities, $p(s' \mid s, a)$
  - Deterministic transitions:

$$p(s' \mid s, a) = \begin{cases} 1 \text{ if } \delta(s,a) = s' \\ 0 \text{ otherwise} \end{cases}$$

where $\delta(s,a)$ is a transition function

Markov Assumption

$$p(s_{t+1} \mid s_t, a_t, \ldots, s_1, a_1) = p(s_{t+1} \mid s_t, a_t)$$

From the Model

- Policy, $\pi : \mathcal{S} \to \mathcal{A}$
- Value function, $V^{\pi}: \mathcal{S} \to \mathbb{R}$
  - Measures the expected total payoff of starting in some state s and *executing* policy π

Markov Decision Process (MDP)

- For **supervised learning** the **PAC learning framework** provided assumptions about where our data came from:

$$\mathbf{x} \sim p^*(\cdot) \text{ and } y = c^*(\cdot)$$

- For **reinforcement learning** we assume our data comes from a **Markov decision process** (MDP)

Markov Decision Processes (MDP)

In RL, the source of our data is an MDP:

1. Start in some initial state $s_0 \in \mathcal{S}$
2. For time step t:
   1. Agent observes state $s_t \in \mathcal{S}$
   2. Agent takes action $a_t \in \mathcal{A}$ where $a_t = \pi(s_t)$
   3. Agent receives reward $r_t \in \mathbb{R}$ where $r_t = R(s_t, a_t)$
   4. Agent transitions to state $s_{t+1} \in \mathcal{S}$ where $s_{t+1} \sim p(s' \mid s_t, a_t)$
3. Total reward is $\sum_{t=0}^{\infty} \gamma^t r_t$
   - The value γ is the "discount factor", a hyperparameter $0 < \gamma < 1$

- Makes the same Markov assumption we used for HMMs! The next state only depends on the current state and action.
- *Def.*: we **execute** a policy π by taking action $a = \pi(s)$ when in state s

RL: Objective Function

- Goal: Find a policy $\pi : \mathcal{S} \rightarrow \mathcal{A}$ for choosing "good" actions that maximize:

$$\mathbb{E}[\text{total reward}] = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t r_t\right]$$

- The above is called the
  "infinite horizon expected future discounted reward"

RL: Optimal Value Function & Policy

- Bellman Equations:

- Optimal policy:
  - Given V^*, $R(s,a)$, $p(s' \mid s,a), \gamma$ we can compute this!

- Optimal value function:
  - System of $|\mathcal{S}|$ equations and $|\mathcal{S}|$ variables (each variable is some $V^*(s)$ for some state s)
  - Can be written without π^*

EXPLORATION VS. EXPLOITATION

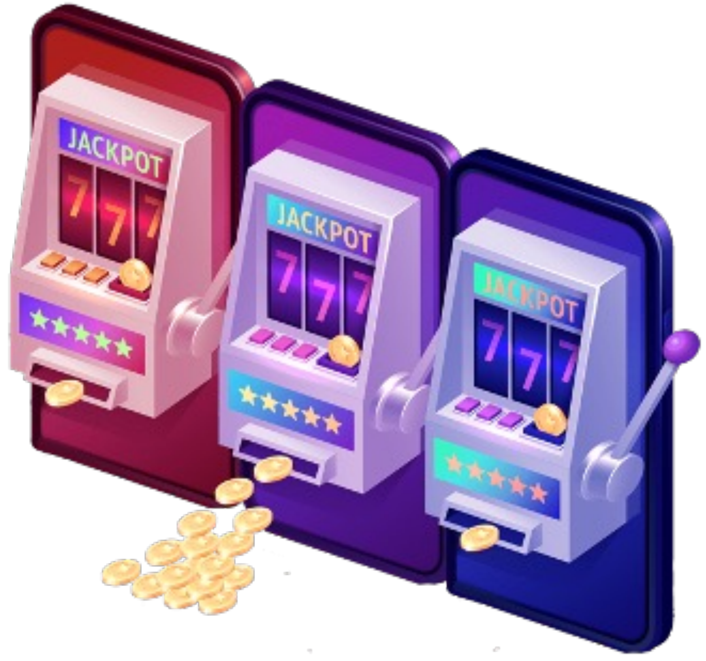
MDP Example: Multi-armed bandit

Single state: $|\mathcal{S}| = 1$

Three actions: $\mathcal{A} = \{1, 2, 3\}$

Deterministic transitions

Rewards are stochastic



Bandit 1	Bandit 2	Bandit 3
???	???	???
???	???	???
???	???	???
???	???	???
???	???	???
???	???	???
???	???	???
???	???	???
???	???	???
???	???	???
???	???	???
???	???	???

Exploration vs. Exploitation Tradeoff

- In RL, there is a **tension** between two strategies an agent can follow when interacting with its environment:
  - **Exploration**: the agent takes actions to visit (state, action) pairs it has not seen before, with the hope of uncovering previously unseen high reward states
  - **Exploitation**: the agent takes actions to visit (state, action) pairs it knows to have high reward, with the goal of maximizing reward given its current (possibly limited) knowledge of the environment
- Balancing these two is critical to success in RL!
  - If the agent **only explores**, it performs no better than a random policy
  - If the agent **only exploits**, it will likely never discover an optimal policy
- One approach for trading off between these:
  the ϵ-greedy policy

FIXED POINT ITERATION

Fixed Point Iteration

$$f_1(x_1, \ldots, x_n) = 0$$
$$\vdots$$
$$f_n(x_1, \ldots, x_n) = 0$$

$$x_1 = g_1(x_1, \ldots, x_n)$$
$$\vdots$$
$$x_n = g_n(x_1, \ldots, x_n)$$

$$x_1^{(t+1)} = g_1(x_1^{(t)}, \ldots, x_n^{(t)})$$
$$\vdots$$
$$x_n^{(t+1)} = g_n(x_1^{(t)}, \ldots, x_n^{(t)})$$

- Fixed point iteration is a general tool for solving systems of equations
- Under the right conditions, it will converge

1. Assume we have n equations and n variables, written f(**x**) = 0 where **x** is a vector
2. Rearrange the equations s.t. each variable x_i has one equation where it is isolated on the LHS
3. Initialize the parameters.
4. For i in {1,…,n}, update each parameter and increment *t*:
5. Repeat #5 until convergence

Fixed Point Iteration

$$\cos(y) - x = 0$$
$$\sin(x) - y = 0$$

$$x = \cos(y)$$
$$y = \sin(x)$$

$$x^{(t+1)} = \cos(y^{(t)})$$
$$y^{(t+1)} = \sin(x^{(t)})$$

- Fixed point iteration is a general tool for solving systems of equations
- Under the right conditions, it will converge

1. Assume we have n equations and n variables, written f(**x**) = 0 where **x** is a vector
2. Rearrange the equations s.t. each variable x_i has one equation where it is isolated on the LHS
3. Initialize the parameters.
4. For i in {1,…,n}, update each parameter and increment *t*:
5. Repeat #5 until convergence

Fixed Point Iteration

$$\cos(y) - x = 0$$
$$\sin(x) - y = 0$$

$$x = \cos(y)$$
$$y = \sin(x)$$

$$x^{(t+1)} = \cos(y^{(t)})$$
$$y^{(t+1)} = \sin(x^{(t)})$$

We can implement our example in a few lines of code

```
from math import *

def f(x, y):
    eq1 = cos(y) - x
    eq2 = sin(x) - y
    return (eq1, eq2)

def g(x, y):
    x = cos(y)
    y = sin(x)
    return (x, y)

def fpi(x0, y0, n):
    '''Solves the system of equations by fixed point iteration
    starting at x0 and stopping after n iterations. Also
    includes an auxiliary function f to test at each value.'''
    x = x0
    y = y0
    for i in range(n):
        ox, oy = f(x,y)
        print("i=%2d x=%.4f y=%.4f f(x,y)=(%.4f, %.4f)" % (i, x, y, ox, oy))
        x,y = g(x,y)
    i += 1
    print("i=%2d x=%.4f y=%.4f f(x,y)=(%.4f, %.4f)" % (i, x, y, ox, oy))
    return x,y

if __name__ == "__main__":
    x,y = fpi(-1, -1, 20)
```

Fixed Point Iteration

```
$ python fixed-point-iteration.py
i= 0 x=-1.0000 y=-1.000 f(x,y)=(1.5403, 0.1585)
i= 1 x=0.5403 y=0.5144 f(x,y)=(0.3303, 0.0000)
i= 2 x=0.8706 y=0.7647 f(x,y)=(-0.1490, 0.0000)
i= 3 x=0.7216 y=0.6606 f(x,y)=(0.0681, 0.0000)
i= 4 x=0.7896 y=0.7101 f(x,y)=(-0.0313, 0.0000)
i= 5 x=0.7583 y=0.6877 f(x,y)=(0.0144, 0.0000)
i= 6 x=0.7727 y=0.6981 f(x,y)=(-0.0066, 0.0000)
i= 7 x=0.7661 y=0.6933 f(x,y)=(0.0031, 0.0000)
i= 8 x=0.7691 y=0.6955 f(x,y)=(-0.0014, 0.0000)
i= 9 x=0.7677 y=0.6945 f(x,y)=(0.0006, 0.0000)
i=10 x=0.7684 y=0.6950 f(x,y)=(-0.0003, 0.0000)
i=11 x=0.7681 y=0.6948 f(x,y)=(0.0001, 0.0000)
i=12 x=0.7682 y=0.6949 f(x,y)=(-0.0001, 0.0000)
i=13 x=0.7681 y=0.6948 f(x,y)=(0.0000, 0.0000)
i=14 x=0.7682 y=0.6948 f(x,y)=(-0.0000, 0.0000)
i=15 x=0.7682 y=0.6948 f(x,y)=(0.0000, 0.0000)
i=16 x=0.7682 y=0.6948 f(x,y)=(-0.0000, 0.0000)
i=17 x=0.7682 y=0.6948 f(x,y)=(0.0000, 0.0000)
i=18 x=0.7682 y=0.6948 f(x,y)=(-0.0000, 0.0000)
i=19 x=0.7682 y=0.6948 f(x,y)=(0.0000, 0.0000)
i=20 x=0.7682 y=0.6948 f(x,y)=(0.0000, 0.0000)
```

We can implement our example in a few lines of code

```python
from math import *

def f(x, y):
    eq1 = cos(y) - x
    eq2 = sin(x) - y
    return (eq1, eq2)

def g(x, y):
    x = cos(y)
    y = sin(x)
    return (x, y)

def fpi(x0, y0, n):
    '''Solves the system of equations by fixed point iteration
    starting at x0 and stopping after n iterations. Also
    includes an auxiliary function f to test at each value.'''
    x = x0
    y = y0
    for i in range(n):
        ox, oy = f(x,y)
        print("i=%2d x=%.4f y=%.4f f(x,y)=(%.4f, %.4f)" % (i, x, y, ox, oy))
        x,y = g(x,y)
    i += 1
    print("i=%2d x=%.4f y=%.4f f(x,y)=(%.4f, %.4f)" % (i, x, y, ox, oy))
    return x,y

if __name__ == "__main__":
    x,y = fpi(-1, -1, 20)
```

VALUE ITERATION

RL Terminology

Question: Match each term (on the left) to the corresponding statement or definition (on the right)

Terms:

A. a reward function
B. a transition probability
C. a policy
D. state/action/reward triples
E. a value function
F. transition function
G. an optimal policy
H. Matt's favorite statement

Statements:

1. gives the expected future discounted reward of a state
2. maps from states to actions
3. quantifies immediate success of agent
4. is a deterministic map from state/action pairs to states
5. quantifies the likelihood of landing a new state, given a state/action pair
6. is the desired output of an RL algorithm
7. can be influenced by trading off between exploitation/exploration

RL: Optimal Value Function & Policy

- Bellman Equations:

$$V^\pi(s) = \quad R(s,\pi(s)) + \gamma \sum_{s' \in \mathcal{S}} p(s' \mid s,\pi(s)) V^\pi(s')$$

- Optimal policy:
  - Given V^*, $R(s,a)$, $p(s' \mid s,a), \gamma$ we can compute this!

$$\pi^*(s) = \operatorname*{argmax}_{a \in \mathcal{A}} \underbrace{R(s,a)}_{\text{Immediate reward}} + \underbrace{\gamma \sum_{s' \in \mathcal{S}} p(s' \mid s,a) V^*(s')}_{\text{(Discounted) Future reward}}$$

- Optimal value function:

$$V^*(s) = \max_{a \in \mathcal{A}} R(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s' \mid s,a) V^*(s')$$

  - System of $|\mathcal{S}|$ equations and $|\mathcal{S}|$ variables (each variable is some $V^*(s)$ for some state s)
  - Can be written without π^*

Example: Path Planning

Value Iteration

Algorithm:

Example:

Value Iteration

Algorithm 1 Value Iteration (deterministic transitions)

1: **procedure** VALUEITERATION($R(s,a)$ reward function, $\delta(s,a)$ transition function)
2: Initialize value function $V(s) = 0$ or randomly
3: **while** not converged **do**
4: **for** $s \in \mathcal{S}$ **do**
5: $V(s) = \max_a R(s,a) + \gamma V(\delta(s,a))$
6: Let $\pi(s) = \text{argmax}_a R(s,a) + \gamma V(\delta(s,a)), \ \forall s$
7: **return** π

Variant 1: without Q(s,a) table

Value Iteration

Algorithm 1 Value Iteration (stochastic transitions)

procedure VALUEITERATION($R(s,a)$ reward function, $p(\cdot|s,a)$ transition probabilities)
    Initialize value function $V(s) = 0$ or randomly
    while not converged **do**
        for $s \in \mathcal{S}$ **do**
            $V(s) = \max_a R(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a)V(s')$
    Let $\pi(s) = \operatorname{argmax}_a R(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a)V(s'), \ \forall s$
    return π

Variant 1: without Q(s,a) table

Value Iteration

Algorithm 1 Value Iteration (stochastic transitions)

1: **procedure** VALUEITERATION($R(s,a)$ reward function, $p(\cdot|s,a)$ transition probabilities)
2: Initialize value function $V(s) = 0$ or randomly
3: **while** not converged **do**
4: **for** $s \in \mathcal{S}$ **do**
5: **for** $a \in \mathcal{A}$ **do**
6: $Q(s,a) = R(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) V(s')$
7: $V(s) = \max_a Q(s,a)$
8: Let $\pi(s) = \operatorname{argmax}_a Q(s,a), \ \forall s$
9: **return** π

Variant 2: with Q(s,a) table

Synchronous vs. Asynchronous Value Iteration

Algorithm 1 Asynchronous Value Iteration

```
procedure ASYNCHRONOUSVALUEITERATION(R(s,a), p(·|s,a))
    Initialize value function V(s) = 0 or randomly
    while not converged do
        for s ∈ S do
            V(s) = max_a R(s,a) + γ Σ_{s'∈S} p(s'|s,a) V(s')
    Let π(s) = argmax_a R(s,a) + γ Σ_{s'∈S} p(s'|s,a) V(s'), ∀s
    return π
```

asynchronous updates: compute and update V(s) for each state one at a time

Algorithm 1 Synchronous Value Iteration

```
procedure SYNCHRONOUSVALUEITERATION(R(s,a), p(·|s,a))
    Initialize value function V(s)^(0) = 0 or randomly
    t = 0
    while not converged do
        for s ∈ S do
            V(s)^(t+1) = max_a R(s,a) + γ Σ_{s'∈S} p(s'|s,a) V(s')^(t)
        t = t + 1
    Let π(s) = argmax_a R(s,a) + γ Σ_{s'∈S} p(s'|s,a) V(s'), ∀s
    return π
```

synchronous updates: compute all the fresh values of V(s) from all the stale values of V(s), then update V(s) with fresh values

Value Iteration Convergence

very abridged

Theorem 1 (Bertsekas (1989))
V converges to V^*, if each state is visited infinitely often

Theorem 2 (Williams & Baird (1993))
if $max_s|V^{t+1}(s) - V^t(s)| < \epsilon$
then $max_s|V^{t+1}(s) - V^*(s)| < \frac{2\epsilon\gamma}{1-\gamma}, \forall s$

Theorem 3 (Bertsekas (1987))
greedy policy will be optimal in a finite number of steps (even if not converged to optimal value function!)

Holds for both asynchronous and sychronous updates

Provides reasonable stopping criterion for value iteration

Often greedy policy converges well before the value function