Reinforcement learning
(Chapter 21)
Reinforcement learning

• Regular MDP
  – Given:
    • Transition model $P(s' | s, a)$
    • Reward function $R(s)$
  – Find:
    • Policy $\pi(s)$

• Reinforcement learning
  – Transition model and reward function initially unknown
  – Still need to find the right policy
  – “Learn by doing”
Offline (MDPs) vs. Online (RL)

Offline Solution

Online Learning

Source: Berkeley CS188
Reinforcement learning: Basic scheme

• In each time step:
  – Take some action
  – Observe the outcome of the action: successor state and reward
  – Update some internal representation of the environment and policy
  – If you reach a terminal state, just start over (each pass through the environment is called a *trial*)

• Why is this called reinforcement learning?
Applications of reinforcement learning

- Backgammon

Applications of reinforcement learning

- AlphaGo

https://deepmind.com/research/alphago/
Applications of reinforcement learning

• **Learning a fast gait for Aibos**

---

Policy Gradient Reinforcement Learning for Fast Quadrupedal Locomotion

Nate Kohl and Peter Stone.

Applications of reinforcement learning

• Stanford autonomous helicopter

Pieter Abbeel et al.
Applications of reinforcement learning

• Playing Atari with deep reinforcement learning

[Diagram of DQN algorithm]

[Video]

Applications of reinforcement learning

- **End-to-end training of deep visuomotor policies**

Fig. 1: Our method learns visuomotor policies that directly use camera image observations (left) to set motor torques on a PR2 robot (right).

[Video](#)

Sergey Levine et al., Berkeley
Applications of reinforcement learning

• Object detection

Sequence of attended regions to localize the object

States

Actions

Steps

$t_1$ ... $t_i$ $t_i + 1$ ... $t_n - 1$ $t_n$

Video

J. Caicedo and S. Lazebnik,
Active Object Localization with Deep Reinforcement Learning, ICCV 2015
OpenAI Gym

A toolkit for developing and comparing reinforcement learning algorithms. It supports teaching agents everything from walking to playing games like Pong or Go.

Read the launch blog post ›
View documentation ›
View on GitHub ›

https://gym.openai.com/
Reinforcement learning strategies

• **Model-based**
  – Learn the model of the MDP (transition probabilities and rewards) and try to solve the MDP concurrently

• **Model-free**
  – Learn how to act without explicitly learning the transition probabilities $P(s' \mid s, a)$
  – **Q-learning**: learn an action-utility function $Q(s,a)$ that tells us the value of doing action $a$ in state $s$
Model-based reinforcement learning

• Learning the model:
  – Keep track of how many times state \( s' \) follows state \( s \) when you take action \( a \) and update the transition probability \( P(s' | s, a) \) according to the relative frequencies
  – Keep track of the rewards \( R(s) \)

• Learning how to act:
  – Estimate the utilities \( U(s) \) using Bellman’s equations
  – Choose the action that maximizes expected future utility:

\[
\pi^*(s) = \arg \max_{a \in A(s)} \sum_{s'} P(s'|s,a)U(s')
\]

  – Is there any problem with this “greedy” approach?
Exploration vs. exploitation

Source: Berkeley CS188
Exploration vs. exploitation

- **Exploration**: take a new action with unknown consequences
  - **Pros**:
    - Get a more accurate model of the environment
    - Discover higher-reward states than the ones found so far
  - **Cons**:
    - When you’re exploring, you’re not maximizing your utility
    - Something bad might happen
- **Exploitation**: go with the best strategy found so far
  - **Pros**:
    - Maximize reward as reflected in the current utility estimates
    - Avoid bad stuff
  - **Cons**:
    - Might also prevent you from discovering the true optimal strategy
Exploration strategies

• **Idea**: explore more in the beginning, become more and more greedy over time

• **$\varepsilon$-greedy**: with probability $1-\varepsilon$, follow the greedy policy, with probability $\varepsilon$, take random action
  – Possibly decrease $\varepsilon$ over time

• More complex **exploration functions** to bias towards less visited state-action pairs
  – E.g., keep track of how many times each state-action pair has been seen, return over-optimistic utility estimate if a given pair has not been seen enough times
Model-free reinforcement learning

- **Idea**: learn how to act without explicitly learning the transition probabilities $P(s' \mid s, a)$

- **Q-learning**: learn an *action-utility function* $Q(s,a)$ that tells us the value of doing action $a$ in state $s$
Model-free reinforcement learning

- **Idea:** learn how to act without explicitly learning the transition probabilities $P(s' \mid s, a)$
- **Q-learning:** learn an *action-utility function* $Q(s,a)$ that tells us the value of doing action $a$ in state $s$
- Relationship between Q-values and utilities:

  $$U(s) = \max_a Q(s,a)$$

- With Q-values, you don’t need the transition model to select the next action:

  $$\pi^*(s) = \arg \max_a Q(s,a)$$

- Compare with:

  $$\pi^*(s) = \arg \max_a \sum_{s'} P(s' \mid s,a) U(s')$$
Model-free reinforcement learning

- **Q-learning**: learn an action-utility function $Q(s,a)$ that tells us the value of doing action $a$ in state $s$

$$U(s) = \max_a Q(s,a)$$

- Bellman equation for Q values:

$$Q(s,a) = R(s) + \gamma \sum_{s'} P(s'|s,a) \max_{a'} Q(s',a')$$

- Compare to Bellman equation for utilities:

$$U(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s'|s,a) U(s')$$
Model-free reinforcement learning

- **Q-learning**: learn an action-utility function $Q(s,a)$ that tells us the value of doing action $a$ in state $s$

  $$U(s) = \max_a Q(s,a)$$

- Bellman equation for Q values:

  $$Q(s,a) = R(s) + \gamma \sum_{s'} P(s'|s,a) \max_a Q(s',a')$$

  - Problem: we don’t know (and don’t want to learn) $P(s'|s,a)$
  - Solution: build up estimates of $Q(s,a)$ over time by making small updates based on observed transitions
TD learning

• Motivation: the mean of a sequence $x_1, x_2, \ldots$ can be computed incrementally:

\[
\mu_k = \frac{1}{k} \sum_{i=1}^{k} x_i = \frac{1}{k} \left( x_k + \sum_{i=1}^{k-1} x_i \right)
\]

\[
= \frac{1}{k} (x_k + (k-1)\mu_{k-1}) = \mu_{k-1} + \frac{1}{k} (x_k - \mu_{k-1})
\]

• By analogy, temporal difference (TD) updates to $Q(s,a)$ have the form

\[
Q(s, a) \leftarrow Q(s, a) + \alpha \left( Q^{\text{target}}(s, a) - Q(s, a) \right)
\]

Source: D. Silver
TD learning

• TD update:

\[ Q(s,a) \leftarrow Q(s,a) + \alpha \left( Q^{target}(s,a) - Q(s,a) \right) \]

• Suppose we have observed the transition \((s,a,s')\)

\[ Q^{target}(s,a) = R(s) + \gamma \max_{a'} Q(s',a') \]

• The target is the return if \((s,a,s')\) was the only possible transition:

\[ Q(s,a) = R(s) + \gamma \sum_{s'} P(s'|s,a) \max_{a'} Q(s',a') \]
TD learning

• TD update:

\[ Q(s, a) \leftarrow Q(s, a) + \alpha \left( Q^{\text{target}}(s, a) - Q(s, a) \right) \]

• Suppose we have observed the transition \((s, a, s')\)

\[ Q^{\text{target}}(s, a) = R(s) + \gamma \max_a Q(s', a') \]

• Full update equation:

\[ Q(s, a) \leftarrow Q(s, a) + \alpha \left( R(s) + \gamma \max_{a'} Q(s', a') - Q(s, a) \right) \]

“Updating a guess towards a guess”
TD algorithm outline

• At each time step \( t \)
  – From current state \( s \), select an action \textit{a given exploration policy}
  – Get the successor state \( s' \)
  – Perform the TD update:

\[
Q(s, a) \leftarrow Q(s, a) + \alpha \left( R(s) + \gamma \max_a Q(s', a') - Q(s, a) \right)
\]

\textit{Learning rate}
Should start at 1 and
decay as \( O(1/t) \)
e.g., \( \alpha(t) = c/(c - 1 + t) \)
Exploration policies

• Standard (“greedy”) selection of optimal action:

\[ a = \arg \max_{a'} Q(s, a') \]

• Epsilon-greedy: with probability \( \epsilon \), take random action

• Policy recommended by textbook:

\[ a = \arg \max_{a' \in A(s)} f \left( Q(s, a'), N(s, a') \right) \]

exploration function Number of times we’ve taken action \( a' \) in state \( s \)

\[ f(u, n) = \begin{cases} R^+ & \text{if } n < N_e \text{ (optimistic reward estimate)} \\ u & \text{otherwise} \end{cases} \]
SARSA

- In TD Q-learning, we’re learning about the optimal policy while following the exploration policy

\[ Q(s, a) \leftarrow Q(s, a) + \alpha (R(s) + \gamma \max_{a'} Q(s', a') - Q(s, a)) \]
SARSA

• In TD Q-learning, we’re learning about the optimal policy while following the exploration policy

• Alternative (SARSA): also select action $a'$ according to exploration policy

\[
Q(s, a) \leftarrow Q(s, a) + \alpha \left( R(s) + \gamma Q(s', a') - Q(s, a) \right)
\]

• **SARSA vs. Q-learning example**
TD Q-learning demos

- Andrej Karpathy’s demo
- Older Java-based demo
Function approximation

- So far, we’ve assumed a lookup table representation for utility function $U(s)$ or action-utility function $Q(s,a)$
- But what if the state space is really large or continuous?
- Alternative idea: approximate the utility function, e.g., as a weighted linear combination of features:

$$U(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

  - RL algorithms can be modified to estimate these weights
  - More generally, functions can be nonlinear (e.g., neural networks)

- Recall: features for designing evaluation functions in games
- Benefits:
  - Can handle very large state spaces (games), continuous state spaces (robot control)
  - Can generalize to previously unseen states
Other techniques

• **Policy search**: instead of getting the Q-values right, you simply need to get their ordering right
  – Write down the policy as a function of some parameters and adjust the parameters to improve the expected reward

• **Learning from imitation**: instead of an explicit reward function, you have expert demonstrations of the task to learn from