Intro to reinforcement learning: day 1

Intro to reinforcement learning: day 1

Today, we will introduce the details of RL and some basic algorithms.

Next Tuesday we will extend it more deeply, go over python code which does RL, and discuss the relation to neuroscience, psychology, developmental psychology, and animal behavior.

We will also go over details about the final project next Tuesday.

Objectives - Reinforcement Learning

At the end of the class you should be able to:

- Explain the relationship between decision-theoretic planning (MDPs) and reinforcement learning
- Implement basic state-based reinforcement learning algorithms: Q-learning and SARSA
- Explain the explore-exploit dilemma and solutions
- Explain the difference between on-policy and off-policy reinforcement learning
- Use features for feature-based reinforcement learning

Learning Summary to date

- Given a task, use
  - data/experience
  - bias/background knowledge
  - measure of improvement or error

  to improve performance on the task.
- Representations for:
  - Data (e.g., discrete values, indicator functions)
  - Models (e.g., decision trees, linear functions, linear separators)
- A way to handle overfitting (e.g., trade-off model complexity and fit-to-data, cross validation).
- Search algorithm (usually local, myopic search) to find the best model that fits the data given the bias.

Reinforcement Learning

What should an agent do given:

- **Prior knowledge:** possible states of the world
  possible actions
- **Observations:** current state of world
  immediate reward / punishment
- **Goal:** act to maximize accumulated (discounted) expected reward

Like decision-theoretic planning, except model of dynamics and model of reward not given.

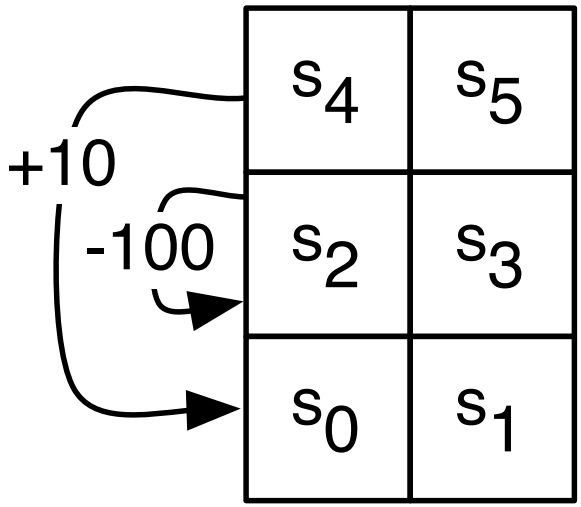
Reinforcement Learning Examples

Often rewards are distant and sparse, only for final outcomes:

- **Game** - reward winning, punish losing
- **Dog** - reward obedience, punish destructive behavior
- **Robot** - reward task completion, punish dangerous behavior

This is closer to general AI... and is the real substance behind the hype of deep learning

RL toy problems



- **upC:** (for "up carefully") Agent goes up, except in states $s4$ and $s5$, where the agent stays still, and has a reward of -1.
- **right:** Agent moves to the right in states $s0, s2, s4$ with a reward of 0 and stays still in other states, with a reward of -1.
- **left:** Agent moves one state to the left in states $s1, s3, s5$. In state s0, it stays in state $s0$ and has a reward of -1. In state s2, it has a reward of -100 and stays in state $s2$. In state $s4$, it gets a reward of 10 and moves to state $s0$.
- **up:** With probability 0.8 it acts like upC, except reward is 0. With probability 0.1 it acts as a left, and with probability 0.1 it acts as right.

Experiences

- We assume there is a sequence of experiences:

  $$\textit{state}, \textit{action}, \textit{reward}, \textit{state}, \textit{action}, \textit{reward}, \ldots.$$

- At any time an agent must decide whether to
  - **explore** to gain more knowledge
  - **exploit** knowledge it has already discovered

Why is reinforcement learning hard?

- What actions are responsible for a reward may have occurred a long time before the reward was received.
- The long-term effect of an action depend on what the agent will do in the future (earlier dependencies or perquisites of future actions).
- The explore-exploit dilemma: at each time should the agent be greedy or inquisitive?

Reinforcement learning: main approaches

- search through a space of policies (a.k.a. controllers)
- learn a model consisting of state transition function $P(s'|a,s)$ and reward function $R(s,a,s')$; solve this an an MDP.
- learn $Q^*(s,a)$, use this to guide action.

Evolutionary Algorithms

- Idea:
  - maintain a population of controllers (policies)
  - evaluate each controller by running it in the environment
  - at each generation, the best controllers are combined to form a new population of controllers
- If there are n states and m actions, there are m^n policies.
- Experiences are used wastefully: only used to judge the whole controller. They don't learn after every step.
- Performance is very sensitive to representation of controller.
- Can occasionally benefit by doing interleaved steps of EA between bouts of classic RL algorithms

Recall: Asynch VI for MDP, storing $Q[s, a]$

(If we knew the model:)

Initialize $Q[S, A]$ arbitrarily
Repeat forever:

- Select state s, action a
- $Q[s, a] \leftarrow$
  $\sum_{s'} P(s'|s, a) \left(R(s, a, s') + \gamma \max_{a'} Q[s', a'] \right)$

Recall: Asynch Value Iteration

1: **Procedure** Asynchronous_Value_Iteration(S,A,P,R)
2: **Inputs**
3: S is the set of all states
4: A is the set of all actions
5: P is state transition function specifying $P(s'|s,a)$
6: R is a reward function $R(s,a,s')$
7: **Output**
8: $\pi[s]$ approximately optimal policy
9: $Q[S,A]$ value function
10: **Local**
11: real array $Q[S,A]$
12: action array $\pi[S]$
13: assign $Q[S,A]$ arbitrarily
14: **repeat**
15: select a state s
16: select an action a
17: $Q[s,a] = \sum_{s'} P(s'|s,a)\ (R(s,a,s') + \gamma max_{a'}\ Q[s',a'])$
18: **until** termination
19: **for each** state s **do**
20: $\pi[s] = argmax_a\ Q[s,a]$
21: **return** π,Q

To store $V[s]$ instead, update with:
$V[s] \leftarrow \max_a \sum_{s'} P(s'|s,a)\,(R(s,a,s') + \gamma V[s'])$
and store: $\pi[s] \leftarrow argmax_a \sum_{s'} P(s'|s,a)\,(R(s,a,s') + \gamma V[s'])$

Reinforcement Learning (Deterministic)

- **flat** or modular or hierarchical
- **explicit states** or features or individuals and relations
- static or finite stage or **indefinite stage or infinite stage**
- **fully observable** or partially observable
- **deterministic** or stochastic dynamics
- goals or **complex preferences**
- **single agent** or multiple agents
- knowledge is given or **knowledge is learned**
- **perfect rationality** or bounded rationality

Deterministic RL

Experiential Asynchronous Value Iteration

initialize $Q[S, A]$ arbitrarily
observe current state s
repeat forever:
    select and carry out an action a
    observe reward r and state s'
    $Q[s, a] \leftarrow r + \gamma \max_{a'} Q[s', a']$
    $s \leftarrow s'$
    end-repeat
        if $||V_k - V_{k-1}|| < \theta$
    for each state s
        $\pi[s] = argmax_a Q[s, a]$
    return π, Q

Note: this is mostly just to illustrate a concept, which will be recycled for model-based methods

Reinforcement Learning (stochastic)

- **flat** or modular or hierarchical
- **explicit states** or features or individuals and relations
- static or finite stage or **indefinite stage or infinite stage**
- **fully observable** or partially observable
- deterministic or **stochastic** dynamics
- goals or **complex preferences**
- **single agent** or multiple agents
- knowledge is given or **knowledge is learned**
- **perfect rationality** or bounded rationality

Temporal Differences

Goal: generate a running mean efficiently
Note: general method

- Suppose we have a sequence of values:

$$v_1, v_2, v_3, \ldots$$

and want a running estimate of the average of the first k values:

$$A_k = \frac{v_1 + \cdots + v_k}{k}$$

Temporal Differences (cont)

- Suppose we know A_{k-1} and a new value v_k arrives:

$$\begin{aligned} A_k &= \frac{v_1 + \cdots + v_{k-1} + v_k}{k} \\ &= \frac{k-1}{k} A_{k-1} + \frac{1}{k} v_k \end{aligned}$$

  Let $\alpha_k = \frac{1}{k}$, then

$$\begin{aligned} A_k &= (1 - \alpha_k) A_{k-1} + \alpha_k v_k \\ &= A_{k-1} + \alpha_k (v_k - A_{k-1}) \end{aligned}$$

  "TD formula"

- Instead, often we use this update with α fixed, and can guarantee convergence to average if

$$\sum_{k=1}^{\infty} \alpha_k = \infty \text{ and } \sum_{k=1}^{\infty} \alpha_k^2 < \infty.$$

Q-learning

- **Idea:** store $Q[State, Action]$; update this as in asynchronous value iteration, but using experience (empirical probabilities and rewards).
- Suppose the agent has an experience $\langle s, a, r, s' \rangle$
- This provides one piece of data to update $Q[s, a]$.
- An experience $\langle s, a, r, s' \rangle$ provides a new estimate for the value of $Q^*(s, a)$:

$$r + \gamma \max_{a'} Q[s', a']$$

which can be used in the TD formula giving:

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

Q-learning

initialize $Q[S, A]$ arbitrarily
observe current state s
repeat forever:
    select and carry out an action a
    observe reward r and state s'
    $Q[s, a] \leftarrow Q[s, a] + \alpha\left(r + \gamma \max_{a'} Q[s', a'] - Q[s, a]\right)$
    $s \leftarrow s'$
    end-repeat
        if $||V_k - V_{k-1}|| < \theta$
    for each state s
        $\pi[s] = argmax_a Q[s, a]$
    return π, Q

Q-learning

controller *Q-learning*(S,A,γ,α)
Inputs
S is a set of states
A is a set of actions
γ the discount
α is the step size
Local
real array $Q[S,A]$
previous state s
previous action a
initialize $Q[S,A]$ arbitrarily
observe current state s
repeat
select and carry out an action a
observe reward r and state s'
$Q[s,a] \leftarrow Q[s,a] + \alpha(r + \gamma \max_{a'} Q[s',a'] - Q[s,a])$
$s \leftarrow s'$
until termination

Properties of Q-learning

- Q-learning converges to an optimal policy, no matter what the agent does, as long as it tries each action in each state enough (**off-policy**)
- But what should the agent do?
  - **exploit**: when in state s, select an action that maximizes $Q[s, a]$
  - **explore**: select another action

Areas to improve on Q-learning?

- It does one "**backup**" between each experience, e.g., increasing the expected value of states farther from the reward, by chaining or percolating Q value backwards. Q value is treated like a reward in this way.
  - Is this appropriate for a robot interacting with the real world?
  - An agent can make better use of the data by
    — doing multi-step backups
    — building a model, and using MDP methods to determine optimal policy.
- It learns separately for each state.

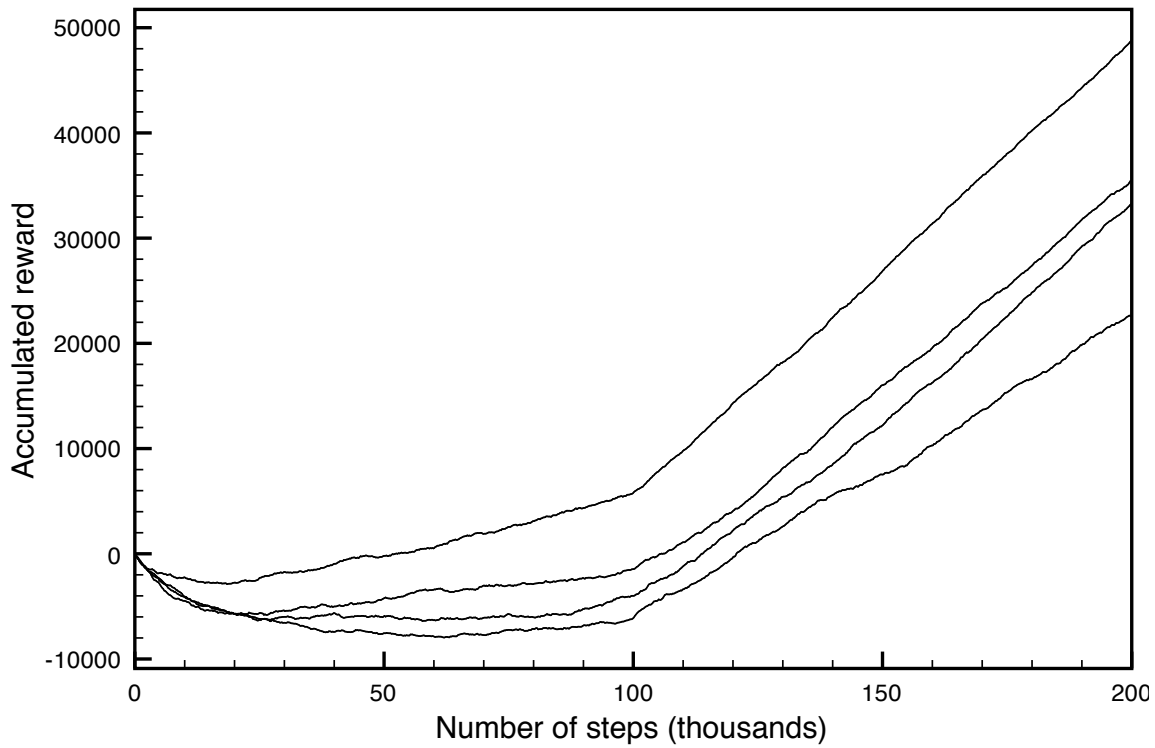
Exploration and Exploitation Strategies

- ϵ-**greedy strategy:** choose a random action with probability ϵ and a best action with probability $1 - \epsilon$.
- **Softmax** action selection: in state s, choose action a with probability increasing for higher Q

$$\frac{e^{Q[s,a]/\tau}}{\sum_a e^{Q[s,a]/\tau}}$$

where $\tau > 0$ is *temperature* defining how much a difference in Q-values maps to probability. Good actions chosen more often than bad actions.
- With either of above, can incorporate a k (time step) value such that exploration decreases over time (often a good idea)
- **"Optimism in the face of uncertainty"** is an alternative solution of initializing Q to high values to encourage exploration (not ideal).

Evaluating RL Algorithms



Is this part of the function (t=0-200) important, or is ultimate online performance the goal? Human babies are dumb and slow-learning compared to goat kid or calf...
Conclusion: which algorithm depends on your problem!

On-policy Learning

- Q-learning does **off-policy learning**: it learns the value of an optimal policy, no matter what it does (given enough exploration).
- This could be bad if the exploration policy is dangerous.
- **On-policy learning** learns the value of the policy being followed:
  e.g., act greedily 80% of the time and act randomly 20% of the time, or don't ever transition from s_1 to s_4, or arbitrarily complete specifications of the policy.
- Why? If the agent is actually going to explore, it may be better to optimize the actual policy it is going to do.
- SARSA uses the experience $\langle s, a, r, s', a' \rangle$ to update $Q[s, a]$.

SARSA (state-action-reward-state-action)

initialize $Q[S, A]$ arbitrarily
observe current state s
select action a using a policy based on Q
repeat forever:
carry out action a
observe reward r and state s'
select action a' using a policy based on Q
$Q[s, a] \leftarrow Q[s, a] + \alpha\left(r + \gamma Q[s', a'] - Q[s, a]\right)$
$s \leftarrow s'$
$a \leftarrow a'$
end-repeat
if $||V_k - V_{k-1}|| < \theta$
for each state s
$\pi[s] = argmax_a Q[s, a]$
return π, Q

Varieties: Can follow an arbitrarily constrained policy

Multi-step backups

Considering updating $Q[s_t, a_r]$ based on "future" experiences:

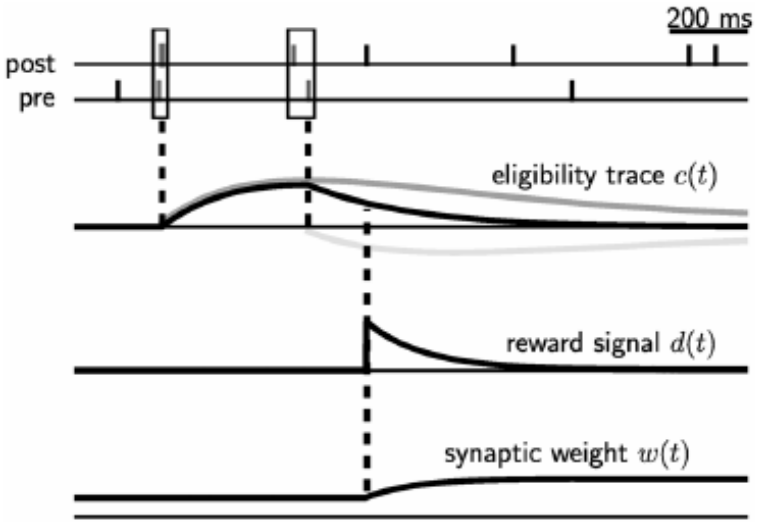
$$s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1}, r_{t+2}, s_{t+2}, a_{t+2}, r_{t+3}, s_{t+3}, a_{t+3}, \ldots$$

- How can an agent use more than one-step lookahead?
- How can we update $Q[s_t, a_t]$ by looking "backwards" from time $t+1$, then at $t+2$, then at $t+3$, etc.?
- backup during training could be called look-ahead during execution.

Multi-step lookaheads (really backups)

lookahead	Weight	Return
1 step	$1-\lambda$	$r_{t+1}+\gamma V(s_{t+1})$
2 step	$(1-\lambda)\lambda$	$r_{t+1}+\gamma r_{t+2}+\gamma^2 V(s_{t+2})$
3 step	$(1-\lambda)\lambda^2$	$r_{t+1}+\gamma r_{t+2}+\gamma^2 r_{t+3}+\gamma^3 V(s_{t+3})$
4 step	$(1-\lambda)\lambda^3$	$r_{t+1}+\gamma r_{t+2}+\gamma^2 r_{t+3}+\gamma^3 r_{t+4}+\gamma^4 V(s_{t+3})$
...	...	...
n step	$(1-\lambda)\lambda^{n-1}$	$r_{t+1}+\gamma r_{t+2}+\gamma^2 r_{t+3}+\cdots+\gamma^n V(s_{t+n})$
...	...	...
total	1	

Can use eligibility trace to weight to change values for actions farther in the past less than those closer to the reward/punishment. We've seen these before:



Model-based Reinforcement Learning

- Above methods were **model-free**
- **Model-based** reinforcement learning uses the experiences in a more efficient manner by learning transitional and reward models
- It is used when collecting experiences is expensive (e.g., in a robot or an online game); an agent can do lots of computation between each experience.
- **Idea:** learn the MDP and interleave acting and planning.
- After each experience, update probabilities and the reward, then do some steps of asynchronous value iteration.
- Similar to the benefit of policy iteration over value iteration

Model-based learner

Data Structures: $Q[S, A]$, $T[S, A, S]$, $C[S, A]$, $R[S, A]$
Assign Q, R arbitrarily, $C = 0$, $T = 0$
observe current state s
repeat forever:
select and carry out action a
observe reward r and state s'
$T[s, a, s'] \leftarrow T[s, a, s'] + 1$
$C[s, a] \leftarrow C[s, a] + 1$
$R[s, a] \leftarrow R[s, a] + (r - R[s, a])/C[s, a]$
$s \leftarrow s'$
repeat for a while:
select state s_1, action a_1

$$Q[s_1, a_1] \leftarrow R[s_1, a_1] + \sum_{s_2} \frac{T[s_1, a_1, s_2]}{C[s_1, a_1]} \left(\gamma \max_{a_2} Q[s_2, a_2] \right)$$

Varieties: Can use a $T[S, A, S]$, $R[S, A, S']$ or $T[S, A]$, $R[S, A]$; C value can be skipped entirely or used for efficiency of computation; only T or R could be stored for a partial model, etc.