On-policy Learning

- Q-learning does **off-policy learning**: it learns the value of an optimal policy, no matter what it does.
- 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
- 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 $Q[s, a]$. 
initialize $Q[S, A]$ arbitrarily
observe current state $s$
select action $a$

repeat forever:
  carry out action $a$
  observe reward $r$ and state $s'$
  select action $a'$ using a policy based on $Q$
  \[ Q[s, a] := Q[s, a] + \alpha \left( r + \gamma Q[s', a'] - Q[s, a] \right) \]
  $s := s'$
  $a := a'$
Q-learning with Action Replay

initialize $Q[S,A]$ arbitrarily
$E = \{\}$
observed current state $s$
select action $a$

repeat forever:
- carry out action $a$
- observe reward $r$ and state $s'$
  $E := E \cup \{(s, a, r, s')\}$
  $Q[s,a] := Q[s,a] + \alpha (r + \gamma \max_{a'} Q[s',a'] - Q[s,a])$

repeat for a while:
  select $(s_1, a_1, r_1, s'_1) \in E$
  $Q[s_1,a_1] := Q[s_1,a_1] + \alpha \left( r_1 + \gamma \max_{a'_1} Q[s'_1,a'_1] - Q[s_1,a_1] \right)$
  $s := s'$
  $a := a'$
Model-based reinforcement learning uses the experiences in a more effective manner.

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.
Model-based learner


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 $s'$

$T[s, a, s'] := T[s, a, s'] + 1$

$C[s, a] := C[s, a] + 1$

$R[s, a] := R[s, a] + (r - R[s, a])/C[s, a]$

repeat for a while:

select state $s_1$, action $a_1$

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

$s := s'$

What goes wrong with this?
Usually we don’t want to reason in terms of states, but in terms of features.

In state-based methods, information about one state cannot be used by similar states.

If there are too many parameters to learn, it takes too long.

Idea: Express the value (Q) function as a function of the features. Most typical is a linear function of the features, or a neural network.
Reinforcement Learning

- 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
SARSA with Generalization

1: controller \texttt{SARSA\_with\_Generalization}(Learner, \gamma)

2: \textbf{Inputs}

3: Learner with operations Learner.add\((x, y)\) and Learner.predict\((x)\).

4: \(\gamma \in [0, 1]\): discount factor

5: observe current state \(s\)

6: select action \(a\)

7: \textbf{repeat}

8: \(\text{do}(a)\)

9: observe reward \(r\) and state \(s'\)

10: select action \(a'\) based on Learner.predict\(((s', a'))\)

11: Learner.add\(((s, a), r + \gamma \ast \text{Learner.predict}((s', a'))))

12: \(s := s'\)

13: \(a := a'\)

14: \textbf{until} termination
Review: Gradient descent

To find a (local) minimum of a real-valued function $f(x)$:

- assign an arbitrary value to $x$
- repeat

$$x := x - \eta \frac{df}{dx}$$

where $\eta$ is the step size

To find a local minimum of real-valued function $f(x_1, \ldots, x_n)$:

- assign arbitrary values to $x_1, \ldots, x_n$
- repeat:
  - for each $x_i$

$$x_i := x_i - \eta \frac{\partial f}{\partial x_i}$$
A linear function of variables $x_1, \ldots, x_n$ is of the form

$$f^\overline{w}(x_1, \ldots, x_n) = w_0 + w_1 x_1 + \cdots + w_n x_n$$

$\overline{w} = \langle w_0, w_1, \ldots, w_n \rangle$ are weights. (Let $x_0 = 1$).

Given a set $E$ of examples.
Example $e$ has input $x_i = e_i$ for each $i$ and observed value, $o_e$:

$$Error_E(\overline{w}) = \sum_{e \in E} (o_e - f^\overline{w}(e_1, \ldots, e_n))^2$$

Minimizing the error using gradient descent, each example should update $w_i$ using:

$$w_i := w_i - \eta \frac{\partial Error_E(\overline{w})}{\partial w_i}$$
Given $E$: set of examples over $n$ features
    each example $e$ has inputs $(e_1, \ldots, e_n)$ and output $o_e$:
Assign weights $\overrightarrow{w} = \langle w_0, \ldots, w_n \rangle$ arbitrarily
repeat:
    For each example $e$ in $E$:
        let $\delta = o_e - f(\overrightarrow{w}(e_1, \ldots, e_n))$
    For each weight $w_i$:
        $w_i := w_i + \eta \delta e_i$
SARSA with linear function approximation

- One step backup provides the examples that can be used in a linear regression.
- Suppose $F_1, \ldots, F_n$ are the features of the state and the action.
- So $Q_w(s, a) = w_0 + w_1 F_1(s, a) + \cdots + w_n F_n(s, a)$
- An experience $\langle s, a, r, s', a' \rangle$ provides the “example”:
  - old predicted value: $Q_w(s, a)$
  - new “observed” value: $r + \gamma Q_w(s', a')$
- Treat $r + \gamma Q_w(s', a')$ as a new training example for $Q(s, a)$ in linear regression (or other supervised learning algorithm).
SARSA with linear function approximation

Given $\gamma$: discount factor; $\eta$: step size
Assign weights $w = \langle w_0, \ldots, w_n \rangle$ arbitrarily
observe current state $s$
select action $a$
repeat forever:
    carry out action $a$
    observe reward $r$ and state $s'$
    select action $a'$ (using a policy based on $Q_w$)
    let $\delta = r + \gamma Q_w(s', a') - Q_w(s, a)$
    For $i = 0$ to $n$
        $w_i := w_i + \eta \delta F_i(s, a)$
    $s := s'$
    $a := a'$
Monster Game

```
0 1 2 3 4

P_3
M
M
M

0

P_1
R

3

M

2

M
M
M

1

0

P_2

4

P_4
```

© 2023 D. L. Poole and A. K. Mackworth
Example Features

- $F_1(s, a) = 1$ if $a$ goes from state $s$ into a monster location and is 0 otherwise.
- $F_2(s, a) = 1$ if $a$ goes into a wall, is 0 otherwise.
- $F_3(s, a) = 1$ if $a$ goes toward a prize.
- $F_4(s, a) = 1$ if the agent is damaged in state $s$ and action $a$ takes it toward the repair station.
- $F_5(s, a) = 1$ if the agent is damaged and action $a$ goes into a monster location.
- $F_6(s, a) = 1$ if the agent is damaged.
- $F_7(s, a) = 1$ if the agent is not damaged.
- $F_8(s, a) = 1$ if the agent is damaged and there is a prize in direction $a$.
- $F_9(s, a) = 1$ if the agent is not damaged and there is a prize in direction $a$. 
Example Features

- $F_{10}(s, a)$ is the distance from the left wall if there is a prize at location $P_0$, and is 0 otherwise.
- $F_{11}(s, a)$ has the value $4 - x$, where $x$ is the horizontal position of state $s$ if there is a prize at location $P_0$; otherwise is 0.
- $F_{12}(s, a)$ to $F_{29}(s, a)$ are like $F_{10}$ and $F_{11}$ for different combinations of the prize location and the distance from each of the four walls.
For the case where the prize is at location $P_0$, the $y$-distance could take into account the wall.
Problems and Variants of function approximation

- This algorithm tends to overfit to current experiences. “Catastrophic forgetting”.
  Solution: remember old \( \langle s, a, r, s' \rangle \) experiences and to carry out some steps of action replay

- Different function approximations, such as
  - a decision tree with a linear function at the leaves (regression tree)
  - a neural network

  could be used, but they require a representation of the states and actions.

- Use the policy to do more than one-step lookahead (better estimate of \( Q(s', a') \)).
  For example, compute expected value by generating samples of the rest of a game.
Evolutionary Algorithms

- In state-based MDPs and reinforcement learning, all local optima are global optima.
- With function approximation, MDP/LR algorithms can get stuck in local optima that can be arbitrarily worse than global optima.
- Evolutionary algorithms can help escape local optima.
- Idea:
  - maintain a population of controllers (e.g., SARSA with function approximation
  - evaluate each controller by running it in the environment
  - at each generation, the best controllers are combined to form a new population of controllers
- Performance is sensitive to representation of controller, and ways to combine them.