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 reward

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

Experiences

We assume there is a sequence of experiences:

- state, action, reward, state, action, reward, ....

What should the agent do next?

It must decide whether to:
- **explore** to gain more knowledge
- **exploit** the knowledge it has already discovered

Reinforcement Learning: “Bandit” problem

Each machine has a $Pr(win)$ ... but you don't know what it is...
Which machine should you play?

Why is reinforcement learning hard?

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

Reinforcement learning: main approaches

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

- 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), and you 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, $T = $ prior counts
$\alpha$ is learning rate
observe current state $s$
repeat forever:
select and carry out action $a$
observe reward $r$ and state $s'$
$T[s, a, s'] \leftarrow T[s, a, s'] + 1$
$R[s, a] \leftarrow \alpha \times r + (1 - \alpha) \times R[s, a]$
repeat for a while (asynchronous VI):
select state $s_1$, action $a_1$
let $P = \sum_{s_2} T[s_1, a_1, s_2]$
$Q[s_1, a_1] \leftarrow \frac{1}{P} \sum_{s_2} T[s_1, a_1, s_2] (R[s_1, a_1] + \gamma \max_{a_2} Q[s_2, a_2])$
$s' \leftarrow s'$

Temporal Differences

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:

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

When a new value $v_k$ arrives:

$A_k = \frac{v_1 + \cdots + v_{k-1} + v_k}{k}$

$kA_k = v_1 + \cdots + v_{k-1} + v_k$

$= (k - 1)A_{k-1} + v_k$

$A_k = \frac{k - 1}{k} A_{k-1} + \frac{1}{k} v_k$

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

$A_k = (1 - \alpha)A_{k-1} + \alpha v_k$

$= A_{k-1} + \alpha (v_k - A_{k-1})$

"TD formula"

Often we use this update with $\alpha$ fixed.

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 $Q[s, a]$.
- The experience $\langle s, a, r, s' \rangle$ provides the data point:

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

which can be used in the TD formula giving:

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

begin
initialize $Q[S, A]$ arbitrarily
observe current state $s$
repeat forever:
select and carry out an action $a$
observe reward $r$ and state $s'$
$Q[s, a] \leftarrow Q[s, a] + \alpha (r + \gamma \max_{a'} Q[s', a'] - Q[s, a])$
$s' \leftarrow s'$;
end-repeat
end
Convergence and Termination

- \( N(s, a) \): number of times action \( a \) taken in state \( s \)
- Learning rate in TD-formula \( \alpha = \frac{1}{N(s,a)} \)
- Q-learning converges if every state-action is visited infinitely often.
  1. \( \sum_{t=0}^{\infty} \alpha_t = \infty \)
  2. \( \sum_{t=0}^{\infty} \alpha_t^2 < \infty \)
- Converge when at time, average error over last \( N \) steps is below a threshold \( \epsilon \):
  \[
  \epsilon > \frac{1}{N} \sum_{i=t-N}^{t} |(r + \gamma \max_{a'} Q(s_{i+1}, a')) - Q(s_i, a)|
  \]

Properties of Q-learning

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

Exploration Strategies

- The \( \epsilon \)-greedy strategy: choose a random action with probability \( \epsilon \) and choose a best action with probability \( 1 - \epsilon \).
- Softmax action selection: in state \( s \), choose action \( a \) with probability
  \[
  \frac{e^{Q[s, a] / \tau}}{\sum_a e^{Q[s, a] / \tau}}
  \]
  where \( \tau > 0 \) is the temperature. Good actions are chosen more often than bad actions; \( \tau \) defines how often good actions are chosen. For \( \tau \to \infty \), all actions are equiprobable. For \( \tau \to 0 \), only the best is chosen.
- "optimism in the face of uncertainty": initialize \( Q \) to values that encourage exploration.
- Upper Confidence Bound (UCB): Also store \( N[s, a] \) (number of times that state-action pair has been tried) and use
  \[
  \arg \max_a \left[ Q(s, a) + k \sqrt{\frac{N[s]}{N[s, a]}} \right]
  \]
  where \( N[s] = \sum_a N[s, a] \)

Example: studentbot

- **studentbot**
  - **state variables**:
    - **tired**: studentbot is tired (no/a bit/very)
    - **passtest**: studentbot passes test (no/yes)
    - **knows**: studentbot’s state of knowledge (nothing/a bit/a lot/everything)
    - **goodtime**: studentbot has a good time (no/yes)
  - **studentbot actions**:
    - **study**: studentbot’s knowledge increases, studentbot gets tired
    - **sleep**: studentbot gets less tired
    - **party**: studentbot has a good time, but gets tired and loses knowledge
    - **take test**: studentbot takes a test (can take test anytime)
Example: studentbot
studentbot
studentbot

studentbot rewards:
- +20 if studentbot passes the test
- +2 if studentbot has a good time when not very tired

basic tradeoff: short term vs. long-term rewards

Studentbot Policy
tired
knows
no
knows
very
a_bit
party
nothing
study
a_bit
take_test
a_lot
everything
nothing
a_bit
sleep

off-policy learning: it learns the value of the optimal policy, no matter what it does.

On-policy learning learns the value of the policy being followed.

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 \( Q[s, a] \).

Large State Spaces

- Computer Go: \( 3^{361} \) states
- Atari Games: \( 210 \times 160 \times 3 \) dimensions (pixels)

\[
Q(w(s, a)) \approx \sum_i w_{ai} x_i
\]

\[
Q_w(s, a) \approx g(x; w)
\]
Recall: Logistic Regression
Logistic function of linear weighted inputs:
\[ \hat{Y}_w(e) = f(w_0 + w_1X_1(e) + \cdots + w_nX_n(e)) = f\left(\sum_{i=0}^{n} w_i X_i(e)\right) \]
The sum of squares error is:
\[ \text{Error}(E, \overline{w}) = \sum_{e \in E} \left[ Y(e) - f\left(\sum_{i=0}^{n} w_i X_i(e)\right) \right]^2 \]
The partial derivative with respect to weight \( w_i \) is:
\[ \frac{\partial \text{Error}(E, \overline{w})}{\partial w_i} = -2 \delta \cdot f'(\sum_{i} w_i X_i(e)) \cdot X_i(e) \]
where \( \delta = (Y(e) - f(\sum_{i} w_i X_i(e))) \).
Thus, each example \( e \) updates each weight \( w_i \) by
\[ w_i \leftarrow w_i + \eta \cdot \delta \cdot f'(\sum_{i} w_i X_i(e)) \cdot X_i(e) \]

SARSA with linear function approximation

Given \( \gamma \): discount factor; \( \alpha \): learning rate
Assign weights \( \overline{w} = \langle w_0, \ldots, w_n \rangle \) arbitrarily

\textbf{begin}
\begin{itemize}
  \item observe current state \( s \)
  \item select action \( a \)
  \item repeat forever:
    \begin{itemize}
      \item carry out action \( a \)
      \item observe reward \( r \) and state \( s' \)
      \item select action \( a' \) (using a policy based on \( Q_w \))
      \item let \( \delta = r + \gamma Q_w(s', a') - Q_w(s, a) \)
      \item For \( i = 0 \) to \( n \)
        \begin{itemize}
          \item \( w_i \leftarrow w_i + \alpha \cdot \delta \cdot \frac{\partial Q_w(s, a)}{\partial w_i} \)
          \item \( s \leftarrow s' \); \( a \leftarrow a' \)
        \end{itemize}
    \end{itemize}
\end{itemize}
\textbf{end-repeat}


Experience Replay

Two tricks used in practice:
1. Experience Replay
2. Use two \( Q \) function (two networks):
   - \( Q \) network (currently being updated)
   - Target network (occasionally updated)

Mitigating Divergence

Experience Replay

Idea: Store previous experiences \( (s, a, r, s', a') \) in a buffer and sample a mini-batch of previous experiences at each step to learn by Q-learning

Breaks correlations between successive updates (more stable learning)

Few interactions with environment needed to converge (greater data efficiency)
Target Network

Idea: use a separate target network that is updated only periodically
- target network has weights $\mathbf{w}$ and computes $Q_{\mathbf{w}}(s, a)$
- repeat for each $(s, a, r, s', a')$ in mini-batch:
  $$\mathbf{w} \leftarrow \mathbf{w} + \alpha \left[ Q_{\mathbf{w}}(s, a) - R(s) - \gamma Q_{\mathbf{w}}(s', a') \right] \frac{\partial Q_{\mathbf{w}}(s, a)}{\partial \mathbf{w}}$$
- $\overline{\mathbf{w}} \leftarrow \mathbf{w}$

Deep Q-Network

Assign weights $\overline{\mathbf{w}} = (w_0, \ldots, w_n)$ at random in $[-1, 1]$

begin
  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 $Q_{\mathbf{w}}$)
    add $(s, a, r, s', a')$ to experience buffer
    Sample mini-batch of experiences from buffer
    For each experience $(\hat{s}, \hat{a}, \hat{r}, \hat{s}', \hat{a}')$ in mini-batch:
      let $\delta = \hat{r} + \gamma Q_{\mathbf{w}}(\hat{s}', \hat{a}') - Q_{\mathbf{w}}(\hat{s}, \hat{a})$
      $\mathbf{w} \leftarrow \mathbf{w} + \alpha \times \delta \times \frac{\partial Q_{\mathbf{w}}(\hat{s}, \hat{a})}{\partial \mathbf{w}}$
    $s \leftarrow s'$; $a \leftarrow a'$;
  every $c$ steps, update target $\overline{\mathbf{w}} \leftarrow \mathbf{w}$
end-repeat
end

Deep Q-Network for Atari


Deep Q-Network vs. Linear Approx.


Mastering GO with MCTS and Neural Networks: AlphaGo


Mastering GO with MCTS and Neural Networks: AlphaGo Zero

Bayesian Reinforcement Learning

- Include the parameters (transition function and observation function) in the state space
- Model-based learning through inference (belief state)
- State space is now continuous, belief space is a space of continuous functions
- Can mitigate complexity by modeling reachable beliefs


Next:

- Affective Computing or POMDP/MCTS/DRL review
- Recap