Reinforcement Learning

Learning from interaction
Learning to achieve goals
Learning about, from, and while interacting with an environment
Learning what to do—how to map situations to actions—so as to maximize cumulative reward

Learning from Interaction with the world

An agent receives sensations or percepts from the environment through its sensors and acts on the environment through its effectors and occasionally receives rewards or punishments from the environment.

The goal of the agent is to maximize its reward (pleasure) or minimize its punishment (or pain) as it stumbles along in an a-priori unknown, uncertain, environment.
Supervised Learning

Experience = Labeled Examples

Inputs ➔ Supervised Learning System ➔ Outputs

Objective – Minimize Error between desired and actual outputs

Reinforcement Learning

Experience = Action-induced State Transitions and Rewards

Inputs ➔ Reinforcement Learning System ➔ Outputs = actions

Objective – Maximize reward

Reinforcement learning

Learner is not told which actions to take
Rewards and punishments may be delayed
  - Sacrifice short-term gains for greater long-term gains
The need to tradeoff between exploration and exploitation
Environment may not be observable or only partially observable
Environment may be deterministic or stochastic
Reinforcement learning

Key elements of an RL System

Policy - what to do
Reward - what is good
Value - what is good because it predicts reward
Model - what follows what

An Extended Example: Tic-Tac-Toe

Assume an imperfect opponent:
—He/she sometimes makes mistakes
A Simple RL Approach to Tic-Tac-Toe

Make a table with one entry per state

<table>
<thead>
<tr>
<th>State</th>
<th>V(s) - estimated probability of winning</th>
</tr>
</thead>
<tbody>
<tr>
<td>✖️</td>
<td>0.5</td>
</tr>
<tr>
<td>✖️</td>
<td>0.5</td>
</tr>
<tr>
<td>✖️</td>
<td>i win</td>
</tr>
<tr>
<td>✖️</td>
<td>b loss</td>
</tr>
<tr>
<td>✖️</td>
<td>0.5 draw</td>
</tr>
</tbody>
</table>

Current state

Possible next states

Pick the next state with the highest estimated prob. of winning — the largest V(s) - a greedy move; Occasionally pick a move at random - an exploratory move.

RL Learning Rule for Tic-Tac-Toe

We increment each V(s) toward V(s') - a backup:

\[
V(s) \leftarrow V(s) + \alpha [V(s') - V(s)]
\]

Why is Tic-Tac-Toe Too Easy?

Number of states is small and finite
One-step look-ahead is always possible
State completely observable
Some Notable RL Applications

- TD-Gammon – world’s best backgammon program (Tesauro)
- Elevator Control – Crites & Barto
- Inventory Management – 10 - 15% improvement over industry standard methods – Van Roy, Bertsekas, Lee and Tsitsiklis
- Dynamic Channel Assignment -- high performance assignment of radio channels to mobile telephone calls – Singh and Bertsekas

The $n$-Armed Bandit Problem

Choose repeatedly from one of $n$ actions; each choice is called a play. After each play, you get a reward $r_t$, where $E[r_t | a_t] = Q^*(a_t)$. Distribution of $r_t$ depends only on $a_t$. Objective is to maximize the reward in the long term, e.g., over 1000 plays.

The Exploration – Exploitation Dilemma

Suppose you form action value estimates $Q_t(a) = Q(a)$. The greedy action at $t$ is $a_t = \text{arg max}_a Q_t(a)$.

You can exploit all the time $a_t = a_t^*$ ⇒ exploitation

You can never stop exploring; but you should always reduce exploring $a_t \neq a_t^*$ ⇒ exploration
Action-Value Methods

Adapt action-value estimates and nothing else.
Suppose by the t-th play, action \( a \) had been chosen \( k \) times, producing rewards \( r_1, r_2, \ldots, r_k \), then

\[
Q_t(a) = \frac{r_1 + r_2 + \cdots + r_k}{k_t}
\]

\[
\lim_{k_t \to \infty} Q_t(a) = Q^*(a)
\]

\-\Greedy Action Selection

Greedy

\[
d_t = a^*_t = \arg \max_a Q_t(a)
\]

\-\Greedy

\[
d_t = a^*_t \text{ with probability } 1 - \varepsilon
\]

\[
d_t = \text{ random action with probability } \varepsilon
\]

Boltzmann

\[
\text{Pr(choosing action } a \text{ at time } t) = \frac{e^{Q_t(a)/\tau}}{\sum_{a'} e^{Q_t(a')/\tau}}
\]

where \( \tau \) is computational temperature

Incremental Implementation

Recall the sample average estimation method
The average of the first \( k \) rewards is

\[
Q_k = \frac{r_1 + r_2 + \cdots + r_k}{k}
\]

Incremental update rule – does not require storing past rewards

\[
Q_{k+1} = Q_k + \frac{1}{k+1}[r_{k+1} - Q_k]
\]
Tracking a Nonstationary Environment

Choosing $Q_k$ to be a sample average is appropriate in a stationary environment in which the dependence of Rewards on actions is time invariant when none of the $Q$ change over time.

In a nonstationary environment, it is better to use exponential, recency-weighted average

$$Q_{k+1} = Q_k + \alpha (r_k - Q_k)$$

for constant $\alpha$, $0 < \alpha \leq 1$

$$= (1-\alpha)Q_k + \sum_0^\infty \alpha(1-\alpha)^i r_i$$

The Agent-Environment Interface

Agent and environment interact at discrete time steps: $t=0,1,2,...$

Agent observes state at step $t$: $s_t \in S$

produces action at step $t$: $a_t \in A(\cdot)$

gets resulting reward: $r_{t+1} \in \mathbb{R}$

and resulting next state: $s_{t+1}$

The Agent Learns a Policy

**Policy at step $t$**, $\pi_t$:

a mapping from states to action probabilities

$$\pi_t(s,a) = \text{probability that } a_t = a \text{ when } s_t = s$$

Reinforcement learning methods specify how the agent changes its policy as a result of experience.

Roughly, the agent’s goal is to get as much reward as it can over the long run.
Agent-Environment Interface -- Goals and Rewards

Is a scalar reward signal an adequate notion of a goal? - maybe not, but it is surprisingly flexible.
A goal should specify what we want to achieve, not how we want to achieve it.
A goal is typically outside the agent’s direct control.
The agent must be able to measure success:
• explicitly
• frequently during its lifespan

Rewards

Suppose the sequence of rewards after step \( t \) is:
\[ r_{t+1}, r_{t+2}, r_{t+3}, \ldots \]
What do we want to maximize?
In general, we want to maximize the expected return, \( E[r_t] \) for each step \( t \).

Episodic tasks - interaction breaks naturally into episodes, e.g., plays of a game, trips through a maze.
\[ R_T = r_{t+1} + r_{t+2} + \ldots + r_T \]
where \( T \) is a final time step at which a terminal state is reached, ending an episode.

Rewards for Continuing Tasks

Continuing tasks: interaction does not have natural episodes.
Discounted return:
\[ R_c = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots = \sum_{k=0}^{\infty} \gamma^k r_{t+k} \]
where \( \gamma \in [0, 1] \) is the discount rate.

shortsighted: \( 0 \leftarrow \gamma \rightarrow 1 \) farsighted
Example – Pole Balancing Task

Avoid failure: the pole falling beyond a critical angle or the cart hitting end of track.

- As an episodic task where episode ends upon failure:
  - reward = +1 for each step before failure
  - return = number of steps before failure

- As a continuing task with discounted return:
  - reward = −1 upon failure; 0 otherwise
  - return = −γ^k, for k steps before failure

In either case, return is maximized by avoiding failure for as long as possible.

Example – Driving task

Get to the top of the hill as quickly as possible.

- reward = −1 for each step where not at top of hill
  - return = −number of steps before reaching top of hill

Return is maximized by minimizing the number of steps taken to reach the top of the hill.

Notation

In episodic tasks, we number the time steps of each episode starting from zero. We usually do not have distinguish between episodes, so we write instead of for the state at step t of episode j.

Think of each episode as ending in an absorbing state that always produces reward of zero:

We can cover all cases by writing

\[ R = \sum_{t=0}^{\infty} \gamma^t r_{t+1} \]
The Markov Property

By the state at step $t$, we mean whatever information is available to the agent at step $t$ about its environment. The state can include immediate sensations, highly processed sensations, and structures built up over time from sequences of sensations. Ideally, a state should summarize past sensations so as to retain all essential information - it should have the Markov Property:

$$\Pr[\omega_t = s', \omega_{t-1} = s, \omega_{t-2} = s_{t-2}, \ldots, \omega_0 = s_0 | \alpha_t = a, \alpha_{t-1} = a_{t-1}, \ldots, \alpha_0 = a_0] = \Pr[\omega_t = s' | s, a].$$

Markov Decision Processes

If a reinforcement learning task has the Markov Property, it is basically a Markov Decision Process (MDP). If state and action sets are finite, it is a finite MDP. To define a finite MDP, you need to specify:

- state and action sets
- one-step dynamics defined by transition probabilities:
  $$\Pr[\omega_t = s', \omega_{t-1} = s, \omega_{t-2} = s_{t-2}, \ldots, \omega_0 = s_0 | \alpha_t = a, \alpha_{t-1} = a_{t-1}, \ldots, \alpha_0 = a_0] = \Pr[\omega_t = s' | s, a].$$
  $$\Pr[\omega_t = s' | s, a],$$
  $$\forall s, s' \in S, \forall a \in A(s).$$

- reward:
  $$\mathbb{E}_\omega[\sum_{t=0}^\infty \gamma^t r_t | \alpha_0 = a_0, \ldots, \alpha_T = a_T],$$
  $$\forall s, s' \in S, \forall a \in A(s).$$

An Example Finite MDP

Recycling Robot

At each step, robot has to decide whether it should (1) actively search for a can, (2) wait for someone to bring it a can, or (3) go to home base and recharge. Searching is better but runs down the battery; if runs out of power while searching, has to be rescued (which is bad). Decisions made on basis of current energy level: high, low.

Reward = number of cans collected
The value of a state is the expected return starting from that state; depends on the agent’s policy:

State-value function for policy π:

\[ \mathbb{E}_\pi \left[ \sum_{t=0}^{\infty} r_{t+1} | s_t = s \right] = \gamma \sum_{t=0}^{\infty} \mathbb{E}_\pi \left[ r_{t+1} | s_t = s \right] \]

The value of taking an action in a state under policy π is the expected return starting from that state, taking that action, and thereafter following π:

Action-value function for policy π:

\[ \mathbb{E}_\pi \left[ \sum_{t=0}^{\infty} r_{t+1} | s_t = s, a = a \right] = \gamma \sum_{t=0}^{\infty} \mathbb{E}_\pi \left[ r_{t+1} | s_t = s, a = a \right] \]

Bellman Equation for a Policy π

The basic idea:

\[ R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots = r_{t+1} + \gamma \sum_{i=0}^{\infty} \gamma^i r_{t+i} \]

So:

\[ V^\pi(s) = \mathbb{E}_\pi \left[ \sum_{t=0}^{\infty} r_t | s_0 = s \right] = \mathbb{E}_\pi \left[ r_0 + \gamma V^\pi(s_1) \right] \]

Or, without the expectation operator:

\[ V^\pi(s) = \sum \pi(s, a) \sum_{s'} P(s'|s, a) \left[ r(s, a) + \gamma V^\pi(s') \right] \]

Bellman Equation

\[ V^\pi(s) = \sum \pi(s, a) \sum_{s'} P(s'|s, a) \left[ r(s, a) + \gamma V^\pi(s') \right] \]

This is a set of equations (in fact, linear), one for each state. The value function for π is its unique solution.

Backup diagrams:
Golf

State is ball location
Reward of -1 for each stroke until the ball is in the hole
Value of a state?

Actions:
- putt (use putter)
- driver (use driver)
putt succeeds anywhere on the green

Optimal Value Functions

For finite MDPs, policies can be partially ordered:

\[ \pi \geq \pi' \text{ if and only if } V^\pi(s) \geq V'^\pi(s) \text{ for all } s \in S \]

There is always at least one (and possibly many) policies that is better than or equal to all the others. This is an optimal policy. We denote them all \( \pi^* \).

Optimal policies share the same optimal state-value function:

\[ V^*(s) = \max_\pi V^\pi(s) \text{ for all } s \in S \]

Optimal policies also share the same optimal action-value function:

\[ Q^*(s,a) = \max_\pi Q^\pi(s,a) \text{ for all } s \in S \text{ and } a \in A(s) \]

This is the expected return for taking action \( a \) in state \( s \) and thereafter following an optimal policy.

Optimal Value Function for Golf

We can hit the ball farther with driver than with putter, but with less accuracy.

\[ Q^*(s, \text{driver}) \] gives the value of using driver first, then using whichever actions are best.
Bellman Optimality Equation for $V^*$

The value of a state under an optimal policy must equal the expected return for the best action from that state:

$$V^*(s) = \max_{a \in A} Q^*(s, a) = \max_{a \in A} \mathbb{E}_T \left[ r_t + \gamma V^*(s_{t+1}) \mid s_t = s, a_t = a \right]$$

The relevant backup diagram:

This is the unique solution of this system of nonlinear equations.

---

Bellman Optimality Equation for $Q^*$

$$Q^*(s, a) = \mathbb{E}_T \left[ r_t + \gamma \max_{a' \in A} Q^*(s_{t+1}, a') \mid s_t = s, a_t = a \right]$$

$$= \sum_{s' \in S} p_{ss'} \left[ r_t + \gamma \max_{a' \in A} Q^*(s', a') \right]$$

The relevant backup diagram:

$Q^*$ is the unique solution of this system of nonlinear equations.

---

Why Optimal State-Value Functions are Useful

Any policy that is greedy with respect to $V^*$ is an optimal policy.

Therefore, given $V^*$ one-step-ahead search produces the long-term optimal actions.
What About Optimal Action-Value Functions?

Given $Q^*$, the agent does not even have to do a one-step-ahead search:

$$
\pi^*(s) = \arg \max_{a \in A(s)} Q^*(s, a)
$$

Solving the Bellman Optimality Equation

Finding an optimal policy by solving the Bellman Optimality Equation requires:

- accurate knowledge of environment dynamics;
- enough space and time to do the computation;
- the Markov Property.

How much space and time do we need?

- polynomial in number of states (via dynamic programming methods),
- BUT, number of states is often huge

We usually have to settle for approximations.

Many RL methods can be understood as approximately solving the Bellman Optimality Equation.

Efficiency of DP

To find an optimal policy is polynomial in the number of states...

BUT, the number of states often grows exponentially with the number of state variables

In practice, classical DP can be applied to problems with a few millions of states.

Asynchronous DP can be applied to larger problems, and appropriate for parallel computation.

It is surprisingly easy to come up with MDPs for which DP methods are not practical.
Markov Decision Processes

Assume
finite set of states \( S \)
set of actions \( A \)
at each discrete time agent observes state \( s_t \in S \) and
chooses action \( a_t \in A \)
then receives immediate reward \( r_t \)
and state changes to \( s_{t+1} \)
Markov assumption: \( s_{t+1} = \delta(s_t, a_t) \) and \( r_t = r(s_t, a_t) \)
- i.e., \( r_t \) and \( s_{t+1} \) depend only on current state and
action
- functions \( \delta \) and \( r \) may be nondeterministic
- functions \( \delta \) and \( r \) not necessarily known to agent

Agent’s learning task

Execute actions in environment, observe results, and
learn action policy \( \pi : S \rightarrow A \) that maximizes
\[ E[r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + ...] \]
from any starting state in \( S \)
here \( 0 \leq \gamma < 1 \) is the discount factor for future rewards

Note something new:
Target function is \( \pi : S \rightarrow A \)
but we have no training examples of form \((s, a)\)
training examples are of form \((\langle s, a \rangle, r)\)

Reinforcement learning problem

Goal: learn to choose actions that maximize
\[ r_0 + \gamma r_1 + \gamma^2 r_2 + ... \]
where \( 0 \leq \gamma < 1 \)
Learning An Action-Value Function

Estimate $Q^*$ for the current behavior policy $\pi$:

$$Q(s, a) \leftarrow Q(s, a) + \alpha (r + \gamma Q(s', a') - Q(s, a))$$

After every transition from a nonterminal state $s$, do:

If $s_{t+1}$ is terminal, then $Q(s, a) = 0$.

Value function

To begin, consider deterministic worlds...

For each possible policy $\pi$ the agent might adopt, we can define an evaluation function over states

$$V^\pi(s) = r + \gamma \sum_{t=0}^{\infty} \gamma^t r_{t+1}$$

where $r, r_{t+1}$ are generated by following policy $\pi$ starting at state $s$.

Restated, the task is to learn the optimal policy $\pi^*$

$$\pi^* = \arg\max_{\pi} V^\pi(s), (\forall s)$$
What to learn

We might try to have agent learn the evaluation function \( V^* \) (which we write as \( V^* \)). It could then do a look-ahead search to choose best action from any state \( s \) because

\[
\pi^*(s) = \arg \max_a [r(s, a) + \gamma V^*(\delta(s, a))]
\]

A problem:
This works well if agent knows \( \delta: S \times A \rightarrow S, \) and \( r: S \times A \rightarrow \mathbb{R} \).
But when it doesn't, it can't choose actions this way.

Action-Value function – Q function

Define a new function very similar to \( V^* \)

\[
Q(s, a) = r(s, a) + \gamma V^*(\delta(s, a))
\]

If agent learns \( Q \), it can choose optimal action even without knowing \( \delta \! \).

\[
\pi^*(s) = \arg \max_a [r(s, a) + \gamma V^*(\delta(s, a))]
\]

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

\( Q \) is the evaluation function the agent will learn.

Training rule to learn Q

Note \( Q \) and \( V^* \) are closely related:

\[
V^*(s) = \max_a Q(s, a)
\]

Which allows us to write \( Q \) recursively as

\[
Q(s, a) = r(s, a) + \gamma \max_{a'} Q(\delta(s, a'))
\]

Let \( \hat{Q} \) denote learner’s current approximation to \( Q \).
Consider training rule

\[
\hat{Q}(s, a) \leftarrow r + \gamma \max_{a'} \hat{Q}(s', a')
\]

where \( s' \) is the state resulting from applying action \( a \) in state \( s \).
Q-Learning

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

Initialize $Q(s, a)$ arbitrarily
Repeat for each episode:
Initialize $s$
For each step of episode:
Choose $a$ using policy derived from $Q$ (e.g., greedy)
Take action $a$, observe $r, s'$,
Update $Q(s, a)$ as follows:
$s \leftarrow s'$
until $s$ is terminal

Q-Learning for Deterministic Worlds

For each $s$, initialize table entry
Observe current state $s$
Do forever:
Select an action $a$ and execute it
Receive immediate reward $r$
Observe the new state $s'$
Update the table entry for $Q(s, a)$ by:
$s' \leftarrow s'$

$$Q(s, a) \leftarrow 0$$

Notice if rewards non-negative, then
and

$$\forall s, a, n : Q_{t+1}(s, a) = Q_t(s, a)$$

$$\forall s, a, n : 0 \leq Q_t(s, a) \leq Q(s, a)$$

Updating $Q$

![Diagram of Q-Learning application](image-url)
Theorem \(\hat{Q}\) converges to \(Q\). Consider case of deterministic world, with bounded immediate rewards, where each \((s, a)\) visited infinitely often.

Proof: Define a full interval to be an interval during which each \((s, a)\) is visited. During each full interval the largest error in \(Q\) table is reduced by factor of \(\gamma\).

Let \(Q\) be table after \(n\) updates, and \(\Delta_n\) be the maximum error in \(Q\); that is
\[
\Delta_n = \max_{s, a} |\hat{Q}(s, a) - Q(s, a)|
\]

For any table entry \(Q(s, a)\) updated on iteration \(n + 1\), the error in the revised estimate \(\hat{Q}(s, a)\) is
\[
|Q_{n+1}(s, a) - Q(s, a)| = |(r + \gamma \max_{a'} Q(s', a')) - (r + \gamma \max_{a'} \hat{Q}(s', a'))|
\]
\[
= \gamma |\max_{a'} Q(s', a') - \max_{a'} \hat{Q}(s', a')|
\]
\[
\leq \gamma \max_{a'} |Q(s', a') - \hat{Q}(s', a')|
\]
\[
\leq \gamma |\max_{a'} (s', a') - Q(s', a')|
\]
\[
|Q_{n+1}(s, a) - Q(s, a)| = \gamma \Delta_n
\]

Note we used general fact that:
\[
\max_{a} f_1(a) - \max_{a} f_2(a) \leq \max_{a} |f_1(a) - f_2(a)|
\]
Non-deterministic case

What if reward and next state are non-deterministic? We redefine $V$ and $Q$ by taking expected values.

$$V^*(s) = E[r_t + \gamma P_{s,a}^* r_{t+1} + \gamma^2 r_{t+2} + ...]$$

$$= E \left[ \sum_{s'} P_{s,a} r_{t+1} \right]$$

$$Q(s, a) = E[r(s, a) + \gamma V^*(\delta(s, a))]$$

Nondeterministic case

$Q$ learning generalizes to nondeterministic worlds. Alter training rule to

$\hat{Q}(s, a) \leftarrow (1 - \alpha) \hat{Q}(s, a) + \alpha [r + \max_{a'} \hat{Q}(s', a')]$

where

$$\alpha = \frac{1}{1 + \text{visits} (s, a)}$$

Convergence of $\hat{Q}$ to $Q$ can be proved [Watkins and Dayan, 1992]

Sarsa

Turn this into a control method by always updating the policy to be greedy with respect to the current estimate.
Temporal Difference Learning

Temporal Difference (TD) learning methods
Can be used when accurate models of the environment are unavailable – neither state transition function nor reward function are known
Can be extended to work with implicit representations of action-value functions
Are among the most useful reinforcement learning methods

TD Prediction

Policy Evaluation (the prediction problem):
for a given policy $\pi$, compute the state-value function $V^\pi$

The simplest TD method, TD(0):

$$V(s_t) \leftarrow V(s_t) + \alpha [r_{t+1} + \gamma V(s_{t+1}) - V(s_t)]$$

target: an estimate of the return

Simplest TD Method
Dynamic Programming

\[ V(s_t) \leftarrow E_s [r_{t+1} + \gamma V(s_{t+1})] \]

TD Bootstraps and Samples

- Bootstrapping - update involves an estimate
- Sampling - update does not involve an expected value
- DP does not sample
- TD samples

Advantages of TD Learning

- TD methods do not require a model of the environment, only experience
- TD methods can be fully incremental
  - You can learn before knowing the final outcome
  - Less memory
  - Less peak computation
- You can learn without the final outcome
- From incomplete sequences
- TD converge (under certain assumptions to be detailed later)
Optimality of TD(0)

Batch Updating: train completely on a finite amount of data, e.g., train repeatedly on 10 episodes until convergence.
Compute updates according to TD(0), but only update estimates after each complete pass through the data.

For any finite Markov prediction task, under batch updating, TD(0) converges for sufficiently small $\alpha$.

Example – TD-Gammon

Learn to play Backgammon (Tesauro, 1995)
Immediate reward:
+100 if win
-100 if lose
0 for all other states

Trained by playing 1.5 million games against itself.
Now comparable to the best human player.

Temporal difference learning

$Q$ learning: reduce discrepancy between successive $Q$ estimates
One step time difference:

\[
Q'(s, a) = r_t + \gamma \max_a Q(s_{t+1}, a)
\]

Why not two steps?

\[
Q^2(s, a) = r_t + \gamma^2 \max_a Q(s_{t+1}, a)
\]

Or $n$?

\[
Q^n(s, a) = r_t + \gamma^{n-1} \max_a Q(s_{t+n}, a)
\]

Blend all of these:

\[
Q^* (s, a) = (1-\lambda)Q'(s, a) + \lambda Q'(s, a) + \lambda^2 Q'(s, a)
\]
Temporal difference learning

\[ Q'(s,a) = (1-\lambda)Q'(s,a) + \lambda Q'^*(s,a) + \lambda Q'Q(s,a) \]

Equivalent expression:

\[ Q'(s,a) = r + \gamma \left[ (1-\lambda) \max_a Q'(s,a) + \lambda Q'Q(s,a) \right] \]

TD(\lambda) algorithm uses above training rule
Sometimes converges faster than Q learning
converges for learning \( V^* \) for any \( 0 \leq \lambda \leq 1 \) (Dayan, 1992)
Tesauro's TD-Gammon uses this algorithm

Handling Large State Spaces

Replace table with neural net or other function approximator
Virtually any function approximator would work
provided it can be updated in an online fashion

Backups as Training Examples

\[ f(x) \leftarrow f(x) + \alpha \left[ r_{i+1} + \gamma f(x_{i+1}) - f(x) \right] \]

Corresponds to the training example

\([\text{description of } s, r_{i+1} + \gamma f(s_{i+1})]\)

input  target output
Gradient Descent Methods

\[ \hat{\theta}_t = (\theta(1), \theta(2), \ldots, \theta(n)) \]

Assume \( \theta \) is a (sufficiently smooth) differentiable function of \( \hat{\theta}_t \), for all \( s \in S \).

Assume, for now, training examples of this form:

[description of \( s, V^*(s) \)].

---

Gradient Descent

\[ \hat{\theta}_{t+1} = \hat{\theta}_t - \frac{1}{2} \alpha V_{\pi}(\hat{\theta}_t) \]

\[ = \hat{\theta}_t - \frac{1}{2} \alpha \sum_{s \in S} P(s) [V^*(s) - V(s)] \]

\[ = \hat{\theta}_t + \alpha \sum_{s \in S} P(s) [V^*(s) - V(s)] \nabla_v V(s) \]

---

Gradient Descent

Use just the per sample gradient

\[ \hat{\theta}_{t+1} = \hat{\theta}_t - \frac{1}{2} \alpha \nabla_v V(\hat{\theta}_t) \]

\[ = \hat{\theta}_t + \alpha \nabla_v [V^*(s) - V(s)] V(s) \]

Since each sample gradient is an unbiased estimate of
the true gradient, this converges to a local minimum of
the MSE if \( \alpha \) decreases appropriately with \( t \).

\[ \mathbb{E} [V^*(s) - V(s)] V(s) = \sum_{s \in S} P(s) [V^*(s) - V(s)] V(s) \]
But We Don’t have these Targets

Suppose we just have targets \( v \), instead:
\[
\theta_{\text{new}} = \theta_{\text{old}} + \alpha (v - V_{\theta}(s)) \nabla_{\theta} V_{\theta}(s)
\]
If each \( v \) is an unbiased estimate of \( V^*(s) \), i.e., \( E[v] = V^*(s) \), then gradient descent converges to a local minimum (provided \( \alpha \) decreases appropriately).

What about TD(\( \lambda \)) Targets?

\[
\theta_{\text{new}} = \theta_{\text{old}} + \alpha (R^s_t - V_{\theta}(s)) \nabla_{\theta} V_{\theta}(s)
\]
\[
\delta_t = r_{t+1} + \gamma V_{\theta}(s_{t+1}) - V_{\theta}(s_t), \text{as usual, and}
\]
\[
\tilde{\delta}_t = \gamma \delta_{t+1} + \nabla_{\theta} V_{\theta}(s_t)
\]

Linear Methods

Represent states as feature vectors:
\[
\phi_s = (\phi_s(1), \phi_s(2), ..., \phi_s(n))^T
\]
\[
V_{\theta}(s) = \theta^T \phi_s = \sum_{i=1}^{n} \theta_i \phi_s(i)
\]
\[
\nabla_{\theta} V_{\theta}(s) = ?
\]
Learning state-action values

Training examples of the form: $(s_t, a_t, r_t)$

The general gradient descent rule:

$$
\theta_{t+1} = \theta_t + \alpha [r_t - Q(s_t, a_t)] \nabla Q(s_t, a_t)
$$

Linear Gradient Descent Watkins’ $Q(\lambda)$

Current research directions

- Use of state abstractions
- Learning from intrinsic rewards
- Learning predictive state representations