Reinforcement Learning. An introduction **Part I**

Cèsar Fernández

Computer Sciences Dpt
Universitat de Lleida

2024

Contents

Reinforcement Learning (RL)

From Sutton and Barto

The idea that we learn by **interacting with our environment** is probably the first to occur to us when we think about the nature of learning. When an infant plays, waves its arms, or looks about, it has no explicit teacher, but it does have a direct sensorimotor connection to its environment

Reinforcement learning is learning what to do –how to map situations to actions– so as to maximize a numerical reward signal. The learner is not told which actions to take, but instead must discover which actions yield the most reward by trying them

Elements of RL

- State: Environment state
- Action: Performed over the environment $\rightarrow$ new state
- Policy: Mapping from states to actions
- Reward: At each time step, environment reacts providing a reward to a given action (what it is good immediately)
- Value function: What is good in the long run. The value of a state is the **expected** reward over future
- Model of the environment: Predicts next state and reward based on current state and action

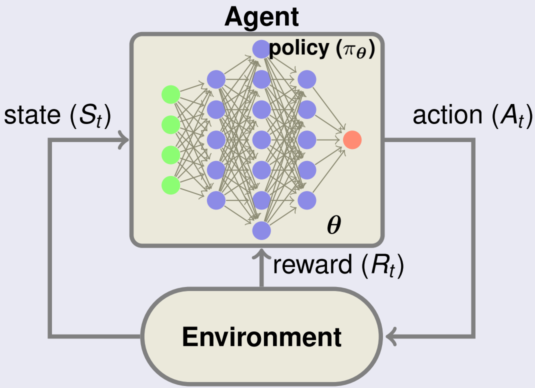
Contents

Finite Markov Decision Process

Finite Markov Decision Process

Abstract framework to model different problems.

Agent-Environment interface



Finite Markov Decision Process

- Trajectory: $S_0, A_0, R_0, S_1, A_1, R_1, \cdots$ (SARSA algorithms)
- Finite: $S_t \in \mathcal{S}, R_t \in \mathcal{R}, A_t \in \mathcal{A}$. Sets: $\mathcal{S}, \mathcal{R}, \mathcal{A}$ are finite
- Markovian:

$$p(s', r|s, a) := Prob\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\}$$

  - No memory process
  - $p()$ is said to be a **model**. Characterizes system dynamics
- State transition probabilities:

$$p(s'|s, a) = \sum_{r \in \mathcal{R}} p(s', r|s, a)$$

- Episode: When reward is accumulated during a finite time interval, it is called an **episode** (aka **trial**)

Finite Markov Decision Process

- Return:

$$G_t := R_{t+1} + R_{t+2} + \cdots + R_T \quad (T : \text{terminal state})$$

when adding a **discount rate** $(0 \leq \gamma \leq 1)$

$$\begin{aligned} G_t &:= R_{t+1} + \gamma \cdot R_{t+2} + \gamma^2 \cdot R_{t+3} + \cdots + \gamma^{T-t-1} \cdot R_T \\ &= \sum_{k=0}^{T-t-1} \gamma^k \cdot R_{t+k+1} \\ &= R_{t+1} + \gamma \cdot G_{t+1} \end{aligned}$$

$(\gamma = 0$, myopic agent)

Finite Markov Decision Process

- Policy (π): Determines $Prob\{A_t|S_t\}$

$$\pi(a|s) := Prob\{A_t = a|S_t = s\}$$

- State value ($v_\pi(s)$): (for a given policy)

$$v_\pi(s) := \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{T-t-1} R_{t+1+k}|S_t = s\right], \forall s \in \mathcal{S}$$

- Q-value ($q_\pi(s, a)$): (or action value)

$$q_\pi(s, a) := \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{T-t-1} R_{t+1+k}|S_t = s, A_t = a\right]$$

v_π and q_π can be estimated from experience

Finite Markov Decision Process

- Bellman equations for $v_\pi(s)$:

$$
\begin{aligned}
v_\pi(s) &:= \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi[R_{t+1} + \gamma \cdot G_{t+1}|S_t = s] \\
&= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a) \left(r + \gamma \cdot \underbrace{\mathbb{E}_\pi[G_{t+1}|S_{t+1} = s']}_{v_\pi(s')} \right) \\
&= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a) \left(r + \gamma \cdot v_\pi(s') \right)
\end{aligned}
$$

Finite Markov Decision Process

- Optimal policies. Optimal values functions:

$$v_*(s) := \max_\pi v_\pi(s)$$

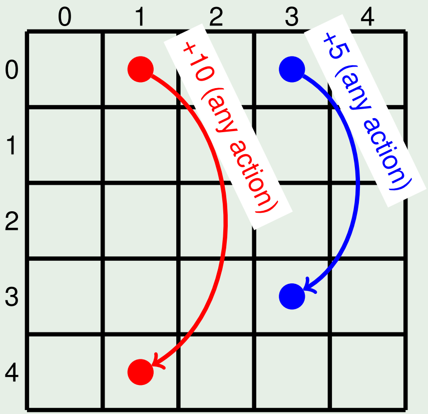
π also shares

$$q_*(s,a) := \max_\pi q_\pi(s,a)$$

$$\begin{aligned} v_*(s) &= \max_{a\in\mathcal{A}(s)} q_{\pi_*}(s,a) = \max_a \mathbb{E}_{\pi_*}[G_t|S_t = s,\ A_t = a] \\ &= \max_a \mathbb{E}_\pi[R_{t+1} + \gamma \cdot G_{t+1}|S_t = s,\ A_t = a] \\ &= \max_a \mathbb{E}_\pi[R_{t+1} + \gamma \cdot v_*(S_{t+1})|S_t = s,\ A_t = a] \\ &= \max_a \sum_{s',r} p(s',r|s,a)\,(r + \gamma \cdot v_*(s')) \end{aligned}$$

Finite Markov Decision Process

Grid world example



- Actions: $\leftrightarrow \updownarrow$
- 0 reward when keep in board
- -1 reward when pushed out (keep in same state)
- $|\mathcal{S}| = 25$, $|\mathcal{A}| = 4$, $\mathcal{R} = \{-1, 0, +5, +10\}$

Finite Markov Decision Process

Grid world example

$$\begin{aligned} v_*(0,0) &= \max_a \sum_{s',r} p(s',r|s,a)\left(r + \gamma \cdot v_*(s')\right) \\ &= \max\left(-1 + \gamma \cdot v_*(0,0), 0 + \gamma \cdot v_*(0,1), 0 + \gamma \cdot v_*(1,0)\right) \end{aligned}$$

$p(s',r|s,a)$ for $s = (0,0)$ only have 4 non zero probs

- $p((0,0),-1|(0,0),\uparrow) = 1$
- $p((0,0),-1|(0,0),\leftarrow) = 1$
- $p((0,1),0|(0,0),\rightarrow) = 1$
- $p((1,0),0|(0,0),\downarrow) = 1$

Solved by a system of inequations with 25 variables

Finite Markov Decision Process

Grid world example

Solving for $\gamma = 0.9$

22.0	24.4	22.0	19.4	17.5
19.8	22.0	19.8	17.8	16.0
17.8	19.8	17.8	16.0	14.4
16.0	17.8	16.0	14.4	13.0
14.4	16.0	14.4	13.0	11.7

v_*

$\rightarrow$	$\uparrow\downarrow\leftarrow\rightarrow$	$\leftarrow$	$\uparrow\downarrow\leftarrow\rightarrow$	$\leftarrow$
$\uparrow\rightarrow$	$\uparrow$	$\leftarrow\uparrow$	$\leftarrow$	$\leftarrow$
$\uparrow\rightarrow$	$\uparrow$	$\leftarrow\uparrow$	$\leftarrow\uparrow$	$\leftarrow\uparrow$
$\uparrow\rightarrow$	$\uparrow$	$\leftarrow\uparrow$	$\leftarrow\uparrow$	$\leftarrow\uparrow$
$\uparrow\rightarrow$	$\uparrow$	$\leftarrow\uparrow$	$\leftarrow\uparrow$	$\leftarrow\uparrow$

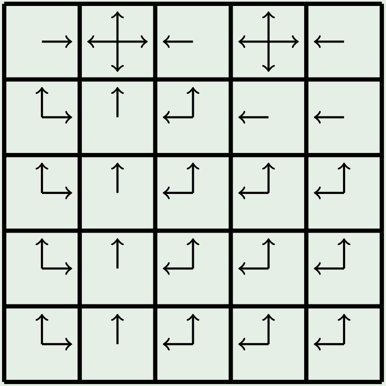
π_*

Finite Markov Decision Process

Grid world example

$v(0,1)$ following optimal policy

22.0	24.4	22.0	19.4	17.5
19.8	22.0	19.8	17.8	16.0
17.8	19.8	17.8	16.0	14.4
16.0	17.8	16.0	14.4	13.0
14.4	16.0	14.4	13.0	11.7



$$v(0,1) = 10 + \gamma \cdot 0 + \gamma^2 \cdot 0 + \gamma^3 \cdot 0 + \gamma^4 \cdot 0 + \gamma^5 \cdot 10 + \cdots = 24.4$$

Contents

Dynamic programming

Definition

Dynamic programming (DP) is the set of algorithms used to compute **optimal policies** for a Markov decision process

Policy evaluation (Prediction)

From Bellman equations, we can:

- Solve the set of equations using classical algebra, or,
- solve iteratively

$$v_\pi(s) := \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a) \left(r + \gamma \cdot v_\pi(s')\right)$$

$$\downarrow$$

$$v_{k+1}(s) = \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a) \left(r + \gamma \cdot v_k(s')\right)$$

Dynamic programming

Policy evaluation (Prediction)

Iterative policy evaluation

π (policy to be evaluated) ;
$V(s) = 0, \forall s \in \mathcal{S}$;
θ (small positive error) ;
repeat
  $\Delta \leftarrow 0$;
  foreach $\underline{s \in \mathcal{S}}$ **do**
    $v \leftarrow V(s)$;
    $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a)\,(r + \gamma \cdot V(s'))$;
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
  end
until $\underline{\Delta < \theta}$;
return $\underline{V \simeq v_\pi}$

Dynamic programming

Policy evaluation (Prediction)

Example `DP-GridWorld.py` evaluates a policy with **same probability for each action** ($\theta = 10^{-5}$)

```
% python DP-GridWorld.py

[[ 3.3  8.8  4.4  5.3  1.5]
 [ 1.5  3.   2.3  1.9  0.5]
 [ 0.1  0.7  0.7  0.4 -0.4]
 [-1.  -0.4 -0.4 -0.6 -1.2]
 [-1.9 -1.3 -1.2 -1.4 -2. ]]
```

$\theta = 10^{-3}$ is enough to compute exactly $v_*(s)$

Dynamic programming

Value iteration (for policy improvement)
From Bellman optimallity Eqs.

$$v_*(s) = \max_a \sum_{s',r} p(s', r|s, a)\,(r + \gamma \cdot v_*(s'))$$

Value iteration

$V(s) = 0,\ \forall s \in \mathcal{S}$;
$\pi \leftarrow$ random policy ;
repeat
    $\Delta \leftarrow 0$;
    foreach $\underline{s \in \mathcal{S}}$ **do**
        $v \leftarrow V(s)$;
        $V(s) \leftarrow \max_a \sum_{s'} \sum_r p(s', r|s, a)\,(r + \gamma \cdot V(s'))$;
        update π with a_* ;
        $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
    end
until $\underline{\Delta < \theta}$;
return $\underline{\pi \simeq \pi_*}$

Dynamic programming

Value iteration (for policy improvement)

Example `DPoptimal-GridWorld.py` gets an optimal policy

```
[[22.  24.4 22.  19.4 17.5]
 [19.8 22.  19.8 17.8 16. ]
 [17.8 19.8 17.8 16.  14.4]
 [16.  17.8 16.  14.4 13. ]
 [14.4 16.  14.4 13.  11.7]]
```

Optimal values, $v_*(s)$

```
→ ↑ ← ↑ ←
↑ ↑ ↑ ← ←
↑ ↑ ↑ ↑ ↑
↑ ↑ ↑ ↑ ↑
↑ ↑ ↑ ↑ ↑
```

Optimal actions, $\pi_*(a|s)$

$\theta = 10^{-3}$ is enough to compute exactly $v_*(s)$

Contents

Monte Carlo methods

- When $|\mathcal{S}|$ is large, DP becomes unfeasible
- Monte Carlo (MC) methods learn from experience: **sampling traces**
  - Create **episodes**
  - **Update** values/policies after each episode

Monte Carlo methods

First visit MC-prediction

π (policy to be evaluated), θ (small positive error) ;
$V(s) = 0, \forall s \in \mathcal{S}$;
$Returns(s) \leftarrow$ empty list $\forall s \in \mathcal{S}$;
repeat
  $Vold(s) \leftarrow V(s), \forall s \in \mathcal{S}$;
  Generate episode following π: $S_0, A_0, R_0, \cdots R_T$;
  $G \leftarrow 0$;
  foreach $0 \leq t \leq T-1$ **do**
    $G \leftarrow G + \gamma \cdot R_{t-1}$;
    if S_t first appearance in Episode **then**
      Append G to $Returns(S_t)$;
      $V(S_t) \leftarrow \mathrm{Average}(Returns(S_t))$
    end
  end
until $\sum_s |V(s) - Vold(s)| < \theta$;
return $V \simeq v_\pi$

Monte Carlo methods

Example `MC-Prediction-GridWorld.py` gets value prediction for a **random policy** with $\theta = 10^{-4}$ and **episode length 10**
After 11,000 episodes:

```
 2.9   9.2   4.0   5.2   1.2
 1.1   2.6   1.7   1.6   0.3
-0.2   0.5   0.4   0.2  -0.4
-0.8  -0.3  -0.2  -0.4  -0.9
-1.4  -1.0  -0.8  -0.9  -1.5
```

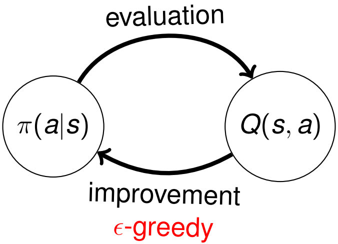
v_π with MC

```
[[ 3.3   8.8   4.4   5.3   1.5]
 [ 1.5   3.    2.3   1.9   0.5]
 [ 0.1   0.7   0.7   0.4  -0.4]
 [-1.   -0.4  -0.4  -0.6  -1.2]
 [-1.9  -1.3  -1.2  -1.4  -2. ]]
```

v_π with DP

Monte Carlo methods

Policy improvement: GPI (Generalized Policy Iteration)



- **On-policy**. The policy being improved is the same as the one being used. ϵ-greedy required
- **Off-policy**. Used and improved policies are decoupled. No ϵ-greedy required. Used policy may be deterministic. As in `DPoptimal-GridWorld.py`

Monte Carlo methods

On-policy First visit MC-control

forall $s \in \mathcal{S},\ a \in \mathcal{A}(s)$ **do**
    $Q(s, a) \leftarrow 0$;
    $Returns(s, a) \leftarrow$ empty list ;
    $\pi(s|a) \leftarrow$ random
end
repeat
    Generate episode following π ;
    foreach (s, a) in episode **do**
        $G \leftarrow$ return of the first occurrence of (s, a) ;
        Append G to $Returns(s, a)$;
        $Q(s, a) \leftarrow \mathrm{Average}(Returns(s, a))$;
    end
    foreach s in episode **do**
        $a_* \leftarrow \arg\max_a Q(s, a)$; ← ties broken randomly
        forall $a \in \mathcal{A}(s)$ **do**
            $$\pi(a|s) \leftarrow \begin{cases} 1 - \epsilon + \epsilon/|\mathcal{A}(s)|, & a = a_* \\ \epsilon/|\mathcal{A}(s)|, & a \neq a_* \end{cases}$$
        end
    end
until forever or if π doesn't change during the last iterations;
return π

note that $\sum_a \pi(a|s) = 1$

Monte Carlo methods

Example `MC-OnPolicy-GridWorld.py` gets optimal policy with $\epsilon = 10^{-1}$ and episode length 100 after 1,000 iterations with same policy

```
[[21.1 23.8 21.4 18.5 16.6]
 [19.  21.4 19.3 16.6 14.9]
 [17.1 19.3 17.4 14.9 13.5]
 [15.4 17.4 15.6 13.5 12.1]
 [13.8 15.6 14.1 12.1 10.9]]
```

v_π with MC

```
→ ↓ ← → ←
→ ↑ ← ↑ ↑
→ ↑ ← ↑ ←
↑ ↑ ↑ ↑ ←
↑ ↑ ↑ ↑ ↑
```

π with MC

```
[[22.  24.4 22.  19.4 17.5]
 [19.8 22.  19.8 17.8 16. ]
 [17.8 19.8 17.8 16.  14.4]
 [16.  17.8 16.  14.4 13. ]
 [14.4 16.  14.4 13.  11.7]]
```

v_π with DP

```
→ ↑ ← ↑ ←
↑ ↑ ↑ ← ←
↑ ↑ ↑ ↑ ↑
↑ ↑ ↑ ↑ ↑
↑ ↑ ↑ ↑ ↑
```

π with DP

Contents

Temporal Difference Learning

Temporal Difference Learning (TD)

TD is a mix of:

- MC: samples the environment
- DP: updates estimates based on learned (**bootstrapping**). Not needed to end the episode

Temporal Difference Learning

Values prediction

- α-MC

$$V(S_t) \leftarrow V(S_t) + \overset{<1,\ \text{step-size parameter}}{\alpha} \cdot (\underbrace{G_t}_{\text{can not be updated until the end of episode}} - V(S_t))$$

- α-TD(0). As

$$G_t = R_{t+1} + \gamma \cdot V(S_{t+1})$$

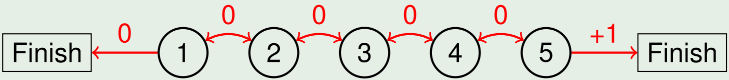
then

$$V(S_t) \leftarrow V(S_t) + \alpha \cdot (\underbrace{R_{t+1} + \gamma \cdot V(S_{t+1})}_{\text{can be updated each time-step}} - V(S_t))$$

TD(0) updates every 1 time-step

Temporal Difference Learning

Random-walk environment

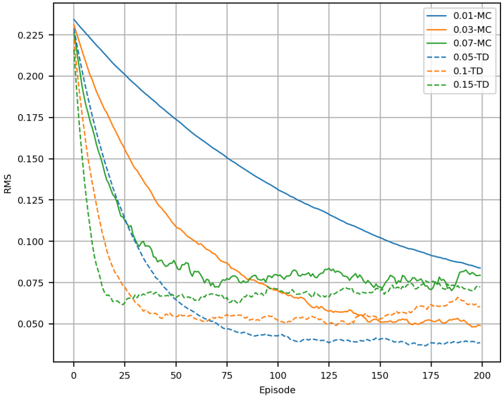


- Number of States (N): 5 ($1 \leq i \leq N$)
- Actions: Go left or right (random)
- Start at state $i = 3$
- Episode ends at Finish
- Reward: +1 when reaching Finish at right. 0 otherwise
- Easily can be proved that: $V(i) = \frac{i}{N+1}$. Solving DP equations ($\gamma = 1$):

$$
\begin{aligned}
v_1 &= 0.5 \cdot v_2 \\
v_2 &= 0.5 \cdot v_1 + 0.5 \cdot v_3 \\
v_3 &= 0.5 \cdot v_2 + 0.5 \cdot v_4 \\
v_4 &= 0.5 \cdot v_3 + 0.5 \cdot v_5 \\
v_5 &= 0.5 \cdot v_4 + 0.5
\end{aligned}
$$

Temporal Difference Learning

Example `alpha-MC-TD0-RandomWalk.py` gets values



- RMS is the root mean square error respect to true values: $V(i) = \frac{i}{N+1}$
- α values: MC={0.01, 0.03, 0.07}, TD={0.05, 0.1, 0.15 }
- Results averaged over 100 repetitions per episode

Temporal Difference Learning

SARSA On-Policy TD Control

Inside an episode, Q-values are updated:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \cdot (R_{t+1} + \gamma \cdot Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t))$$

Temporal Difference Learning

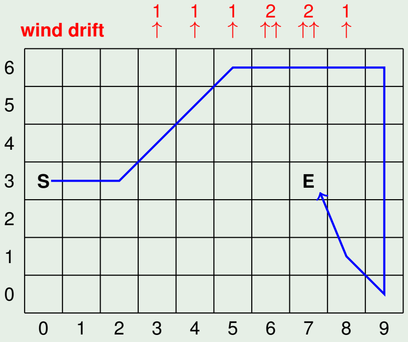
SARSA On-Policy TD Control for estimating Q_*

$Q(s, a) \leftarrow 0$ (or anything) $\forall s \in \mathcal{S},\ a \in \mathcal{A}(s)$;
$Q(terminal\_state, a) \leftarrow 0\ \forall a \in \mathcal{A}(terminal\_state)$;
small ϵ ;
repeat
- S (init state) ;
- Choose A from S following: $\pi(a|s) = \begin{cases} \epsilon, & random(\mathcal{A}(s)) \\ 1-\epsilon, & \arg\max_a Q(s,a) \end{cases}$;
- **repeat**
  - /* steps in episode */ ;
  - Take A. Observe R and S' ;
  - Choose A' from S' following Q (ϵ-greedy);
  - $Q(S, A) \leftarrow Q(S, A) + \alpha \cdot (R + \gamma \cdot Q(S', A') - Q(S', A'))$;
  - $S \leftarrow S'$;
  - $A \leftarrow A'$;
- **until** until S is terminal ;

until for each episode ;

Temporal Difference Learning

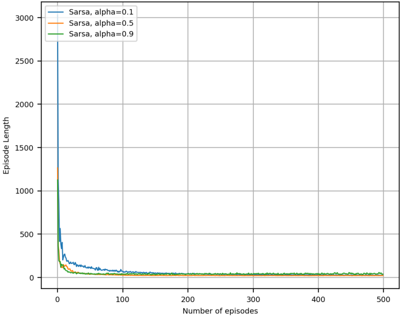
Windy Grid World environment

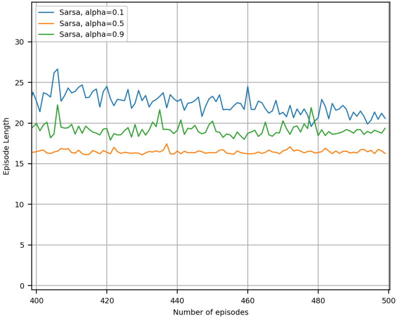


- Actions: [↑,→,↓,←]
- Reward: -1 each move
- $|\mathcal{A}| = 4$
- $|\mathcal{S}| = 10 \cdot 7 = 70$
- **S**: initial state (0,3)
- **E**: terminal state (7,3)
- Objective: find the best policy maximizing reward
- Blue path is optimal. Reward=-17

Temporal Difference Learning

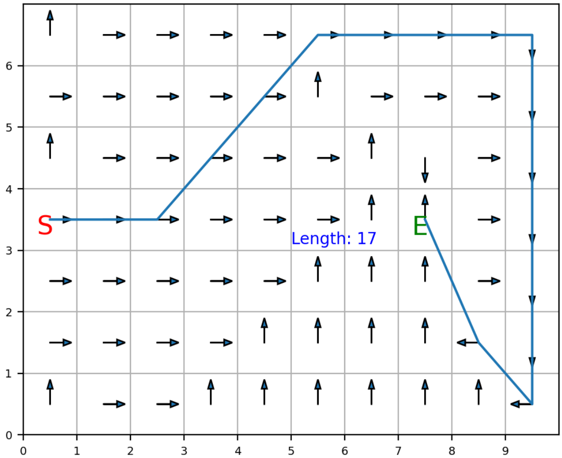
Example `SARSA-WindyGridWorld.py` gets optimal policy with $\epsilon = 0.1$. Averaged for 50 repetitions





Temporal Difference Learning

Example `SARSA-WindyGridWorld.py` gets optimal policy with $\epsilon = 0.1$. Averaged for 50 repetitions



Temporal Difference Learning

SARSA On-Policy TD Control

Inside an episode, Q-values are updated:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \cdot (R_{t+1} + \gamma \cdot Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t))$$

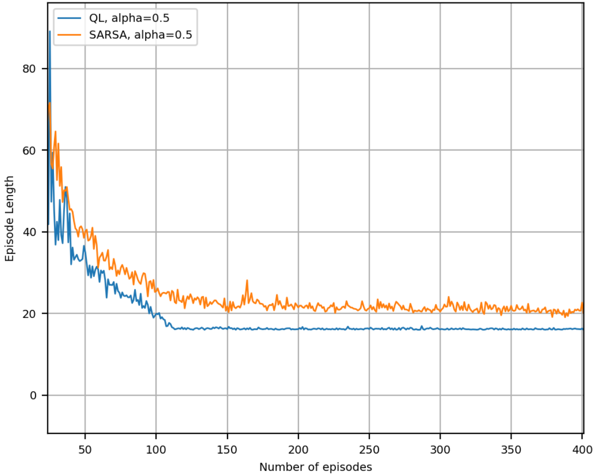
Q-learning Off-policy control

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \cdot \left(R_{t+1} + \gamma \cdot \underline{\max_a Q(S_{t+1}, a)} - Q(S_t, A_t)\right)$$

Q-learning approximates Q_* independently of the current policy

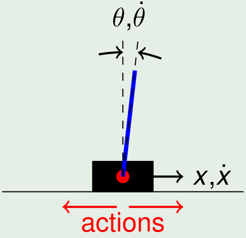
Temporal Difference Learning

Example `QLearning-WindyGridWorld.py` SARSA On-policy and Q-learning Off-policy. $\epsilon = 0.1$ and $\alpha = 0.5$



Temporal Difference Learning

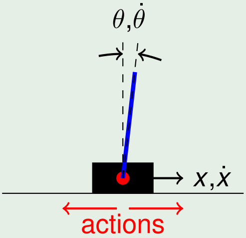
GYM. Cart-Pole environment



- Classic control environment from Gym/Gymnasium project
- Pole attached by an un-actuated joint to a cart. Pendulum upright on the cart
- Goal: Move cart left-right to keep pole upright
- Episode ends when:
  - Cart is outside a margin
  - Pole balances beyond an angle ($|\theta| > 12^o$)
- 4 continuous features define $\mathcal{S}$:
  - Cart position ($-4.8 < x < 4.8$)
  - Cart speed ($-\infty < \dot{x} < \infty$)
  - Pole angle ($-4.18 < \theta < 4.18$)
  - Pole angular speed ($-\infty < \dot{\theta} < \infty$)
- 2 actions: $|\mathcal{A}| = 2$:
- $\mathcal{S}$ must be discretized to build the $Q(s, a)$ table

Temporal Difference Learning

GYM. Cart-Pole environment

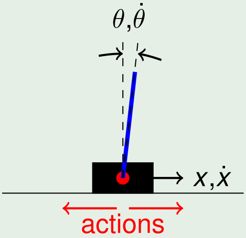


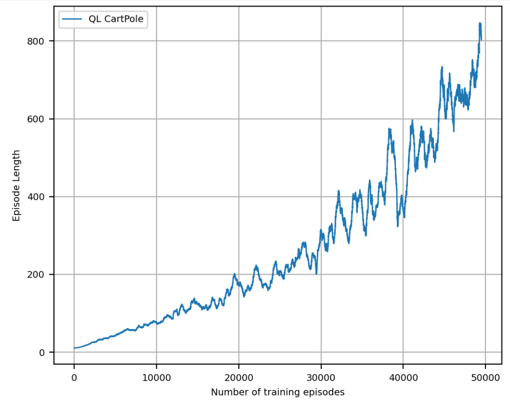
- Effective range considered for $\mathcal{S}$:
  - Cart position ($-1 < x < 1$)
  - Cart speed ($-2 < \dot{x} < 2$)
  - Pole angle ($-1 < \theta < 1$)
  - Pole angular speed ($-2 < \dot{\theta} < 2$)
- Effective range divided into 64 slots
- $|Q| = 64^4 \cdot 2 = 33{,}554{,}423$ possible values
- Only 53,271 states visited after 50,000 episodes
- QL-CartPole.py trains with QL during 50,000 episodes.

  Arguments:
  - `--train`, trains the environment
  - `--render`, shows the environment with Q learnt at last episode
  - No arguments, plots the episode length

Temporal Difference Learning

GYM. Cart-Pole environment





Episode Length averaged with a 500 items moving window

Contents

Approximate Methods

Approximate methods

- As seen, tabular methods ends with large $V(s)/Q(s,a)$ when $\mathcal{S}$ is large
- Instead of learn state values, $V(s)$, or policy, $Q(s,a)$, one can try to approximate them by functions

$$V_\pi(s) \simeq \hat{v}(s,w), \; w \in \mathbb{R}^d$$

  d is the dimension of the weight vector. (i.e. weights of a neural network)
- Prediction objective (mean squared value error) is defined as:

$$\overline{VE}(\mathbf{w}) := \sum_{s\in\mathcal{S}} \mu(s)\left(v_\pi(s) - \hat{v}(s,\mathbf{w})\right)^2$$

  being $\mu(s)$ the probability distribution of $\mathcal{S}$

Approximate Methods

Gradient descent

- Adjust $\mathbf{w}$ in the direction that reduces error $|v_\pi(s) - \hat{v}(s, \mathbf{w})|$

$$
\begin{aligned}
\mathbf{w}_{t+1} &:= \mathbf{w}_t - \frac{1}{2}\alpha \nabla_{\mathbf{w}} \left[v_\pi(s) - \hat{v}(s, \mathbf{w}_t) \right]^2 \\
&= \mathbf{w}_t + \alpha \left[v_\pi(s) - \hat{v}(s, \mathbf{w}_t) \right] \nabla_{\mathbf{w}} \hat{v}(s, \mathbf{w}_t)
\end{aligned}
$$

α: small positive (step-size parameter)

Gradient MC for estimating V_π

π, policy to be evaluated ;
$\hat{v} : \mathcal{S} \times \mathbb{R}^d \to \mathbb{R}$, differentiable function ;
$\mathbf{w} \leftarrow 0$;
repeat
    S (init state) ;
    Generate episode $S_0, A_0, R_1, S_1, A_1, R_2, \cdots, S_T$ with π;
    for $t = 0 \cdots T - 1$ **do**
        $\mathbf{w} \leftarrow \mathbf{w} + \alpha \left[G_t - \hat{v}(S_t, \mathbf{w}) \right] \nabla_{\mathbf{w}} \hat{v}(S_t, \mathbf{w})$
    end
until forever ;

Approximate Methods

Gradient Descent example

Assume we want to find a minimum for $f(x_1, x_2) = x_1^2 + 2 \cdot x_2^2$
By algebra:

$$\nabla f(x_1, x_2) = (2x_1, 4x_2)$$

$$\nabla f(x_1, x_2) = 0, \Rightarrow (x_1 = 0, x_2 = 0)$$

It can be proved that is a minimum

Approximate Methods

Gradient Descent example

Assume we want to find a minimum for $f(x_1, x_2) = x_1^2 + 2 \cdot x_2^2$
By gradient descent:

$$\begin{aligned} (x_{1,t+1}, x_{2,t+1}) &= (x_{1,t}, x_{2,t}) - \frac{\alpha}{2}\nabla f(x_{1,t}, x_{2,t}) \\ &= (x_{1,t}, x_{2,t}) - \frac{\alpha}{2}(2x_{1,t}, 4x_{2,t}) \end{aligned}$$

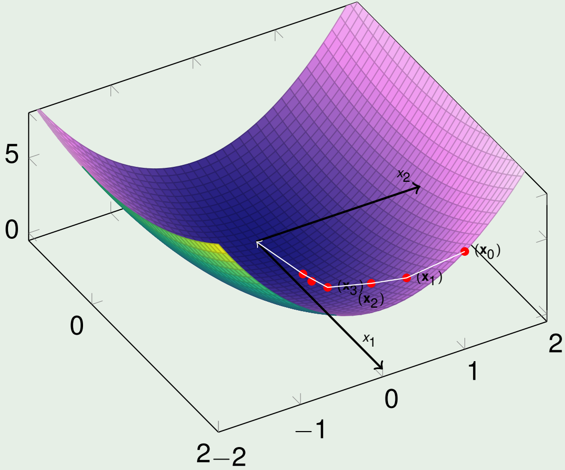
Taking $\alpha = 0.2$ and starting at $(x_{1,0}, x_{2,0}) = (2, 1)$

$$\begin{aligned} (x_{1,1}, x_{2,1}) &= (x_{1,0}, x_{2,0}) - \frac{\alpha}{2}(2x_{1,0}, 4x_{2,0}) \\ &= (2, 1) - 0.1 \cdot (4, 4) = (1.6, 0.6) \\ (x_{1,2}, x_{2,2}) &= (1.6, 0.6) - 0.1 \cdot (3.2, 2.4) = (1.3, 0.4) \\ (x_{1,3}, x_{2,3}) &= (1.3, 0.4) - 0.1 \cdot (2.6, 1.6) = (1, 0.1) \\ (x_{1,4}, x_{2,4}) &= (1, 0.1) - 0.1 \cdot (2, 0.4) = (0.8, 0.06) \end{aligned}$$

Approximate Methods

Gradient Descent example

Assume we want to find a minimum for $f(x_1, x_2) = x_1^2 + 2 \cdot x_2^2$



Approximate Methods

Gradient linear methods

Assume a d-dimensional representation of $v(s)$

$$\hat{v}(s, \mathbf{w}) := \mathbf{w}^\top \mathbf{x}(s) = \sum_{i=1}^{d} w_i x_i(s)$$

Note that:

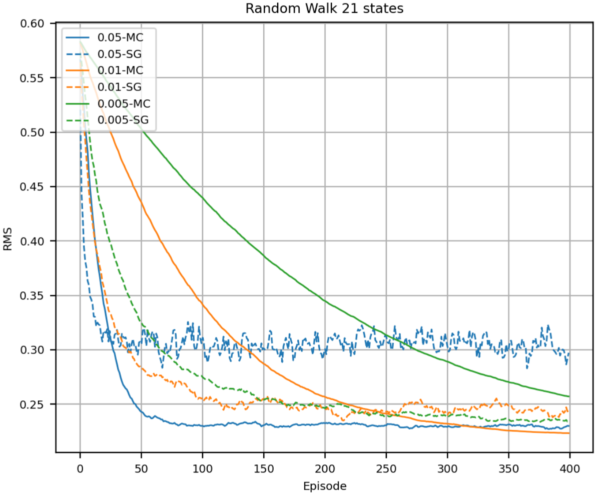
$$\nabla_w \hat{v}(s, \mathbf{w}) = \mathbf{x}(s)$$

$\mathbf{x}(s)$ uses to be any **basis function**. As an example: if $|\mathcal{S}| = 5$:

$$\begin{aligned} \mathbf{x}(S_0) &= [1, 0, 0, 0, 0] \\ \mathbf{x}(S_1) &= [0, 1, 0, 0, 0] \\ &\dots \\ \mathbf{x}(S_4) &= [0, 0, 0, 0, 1] \end{aligned}$$

Approximate Methods

Example `stochastic-gradient-Prediction-RandomWalk.py`
Gradient descent MC for Random Walk Environment with 21 states.
Compared to α-MC



Approximate Methods

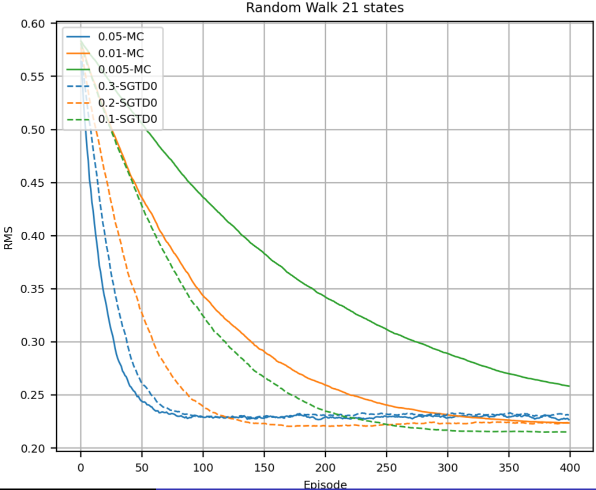
Semi-Gradient TD(0) for estimating V_π

π, policy to be evaluated ;
$\hat{v} : \mathcal{S} \times \mathbb{R}^d \to \mathbb{R}$, differentiable function ;
$\mathbf{w} \leftarrow 0$;
repeat
  S (init state) ;
  repeat
    /* steps in episode */ ;
    Take A. Observe R and S' ;
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha \left[R + \gamma \hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w})\right] \nabla \hat{v}(S, \mathbf{w})$;
    $S \leftarrow S'$;
  until until S is terminal ;
until for each episode ;

Semi-gradient because bootstraps (updates before ending episode).
Could not converge

Approximate Methods

Example stochastic-gradient-Prediction-RandomWalk.py
Gradient descent TD(0) for Random Walk Environment with 21 states. Compared to α-MC



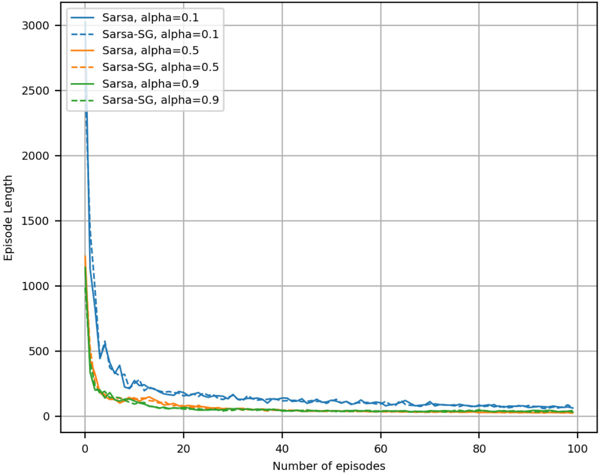
Approximate Methods

Semi-Gradient SARSA for estimating Q

$\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$, differentiable function ;
$\mathbf{w} \leftarrow 0$;
repeat
    S, A (init state and action) ϵ-greedy ;
    repeat
        /* steps in episode */ ;
        Take A. Observe R and S' ;
        if S' is terminal **then**
            $\mathbf{w} \leftarrow \mathbf{w} + \alpha \left[R - \hat{q}(S, A, \mathbf{w})\right] \nabla q(S, A, \mathbf{w})$;
        else
            Choose A' from $\hat{q}(S', \cdot, \mathbf{w})$ (ϵ-greedy) ;
            $\mathbf{w} \leftarrow \mathbf{w} + \alpha \left[R + \gamma \hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})\right] \nabla \hat{q}(S, A, \mathbf{w})$;
            $S \leftarrow S'$;
            $A \leftarrow A'$;
        end
    until until S is terminal ;
until for each episode ;

Approximate Methods

Example `Semigradient-SARSA-WindyGridWorld.py` Semi-Gradient SARSA for Windy Grid World Environment. Compared to SARSA. $\epsilon = 0.1$, averaged on 50 repetitions



Same performance. Not surprising considering that size of Q table and **w** is the same: $|\mathcal{S}| \cdot |\mathcal{A}| = 7 \cdot 10 \cdot 4 = 280$

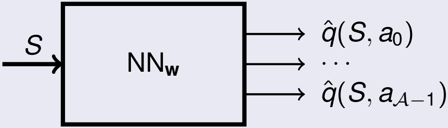
Approximate Methods

Semi-Gradient SARSA with neural nets

- Neural networks perform extremely well as estimating functions
- $\mathbf{w}$ are the weights of the NN layers
- Having:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha \underbrace{[R + \gamma \hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})]}_{\text{NN Loss function}} \underbrace{\nabla \hat{q}(S, A, \mathbf{w})}_{\text{Gradient}}$$

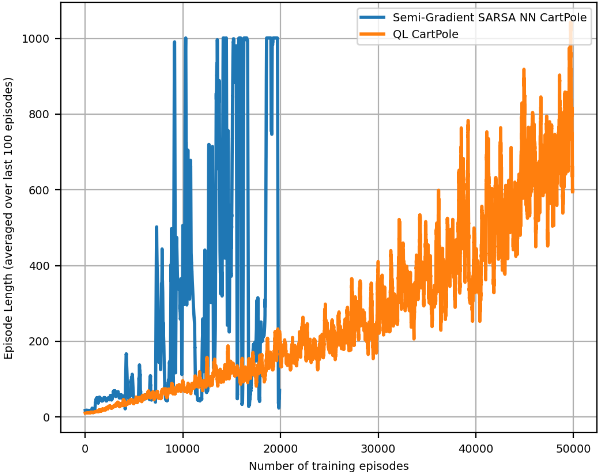
and doing:



- Discretize $\mathcal{S}$ is not required

Approximate Methods

Example `Semigradient-SARSA-NN-Torch-CartPole.py` Semi-Gradient SARSA with NN for Cart-Pole Environment (small network: $1 \times 32 \times 32 \times 2$). Compared to QL



$|\mathbf{w}| = 226$ compared to $|Q| \simeq 33$ millions of QL

Contents

Policy Gradient Methods

Policy Gradient methods

- Don't look at any value function as V or Q
- Instead, for any given policy, $\pi_{\boldsymbol{\theta}}(a|s)$, adjust $\boldsymbol{\theta}$ according some performance measure, $J(\boldsymbol{\theta})$, using gradient approximation:

$$\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t + \alpha \cdot \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t)$$

Policy Gradient Methods

Policy Gradient Theorem

- Helps to find some $J(\boldsymbol{\theta})$ based on policy, $\pi_{\boldsymbol{\theta}}(a|s)$
- Defining, $J(\boldsymbol{\theta}) := v_{\boldsymbol{\theta}}(s_0)$, the PGT proves that:

$$\nabla J(\boldsymbol{\theta}) \propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a|s)$$

where $\mu(s)$ is the probability of being at state s. So,

$$\begin{aligned}
\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) &\propto \mathbf{E}_\pi\left[\sum_a q_\pi(s_t,a) \nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a|s_t)\right] = \mathbf{E}_\pi\left[\sum_a q_\pi(s_t,a) \pi_{\boldsymbol{\theta}}(a|s_t) \frac{\nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a|s_t)}{\pi_{\boldsymbol{\theta}}(a|s_t)}\right] \\
&= \mathbf{E}_\pi\left[q_\pi(s_t,a_t) \frac{\nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a_t|s_t)}{\pi_{\boldsymbol{\theta}}(a_t|s_t)}\right] = \mathbf{E}_\pi\left[G_t \frac{\nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a_t|s_t)}{\pi_{\boldsymbol{\theta}}(a_t|s_t)}\right] \\
&= \mathbf{E}_\pi\left[G_t \nabla_{\boldsymbol{\theta}} \ln \pi_{\boldsymbol{\theta}}(a_t|s_t)\right]
\end{aligned}$$

and using gradient approximation

$$\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t + \alpha \cdot G_t \nabla_{\boldsymbol{\theta}} \ln \pi_{\boldsymbol{\theta}}(a_t|s_t)$$

Policy Gradient Methods

REINFORCE: A MC policy gradient method

$\pi_{\boldsymbol{\theta}}(a|s) : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$, differentiable policy function ;
$\boldsymbol{\theta} \leftarrow 0$;
repeat
    Generate episode: $S_0, A_0, R_1, S_1, \cdots A_{T-1}, R_T$, following $\pi_{\boldsymbol{\theta}}$;
    for $t \in 0 \cdots T-1$ **do**
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha \gamma^t G_t \boldsymbol{\nabla}_{\boldsymbol{\theta}} \ln \pi_{\boldsymbol{\theta}}(a_t|s_t)$;
    end
until forever ;

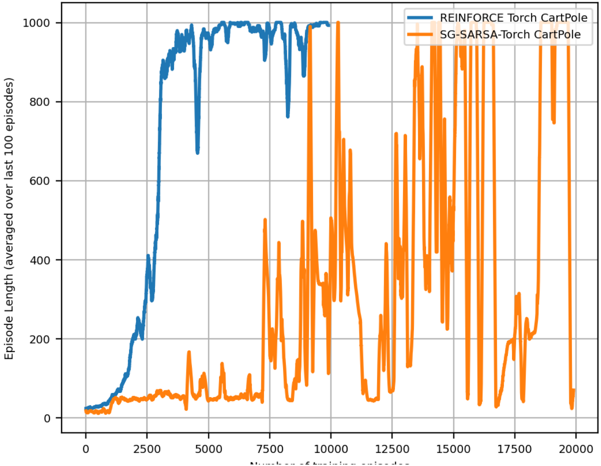
Policy Gradient Methods

REINFORCE: A MC policy gradient method

$\pi_{\boldsymbol{\theta}}(a|s) : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$, differentiable policy function ;
$\boldsymbol{\theta} \leftarrow 0$;
repeat
  Generate episode: $S_0, A_0, R_1, S_1, \cdots A_{T-1}, R_T$, following $\pi_{\boldsymbol{\theta}}$;
  for $t \in 0 \cdots T-1$ **do**
    $$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} \underbrace{+ \alpha\gamma^t G_t \nabla_{\boldsymbol{\theta}} \ln \pi_{\boldsymbol{\theta}}(a_t|s_t)}_{-\alpha\nabla_{\boldsymbol{\theta}}\underbrace{[-\gamma^t G_t \ln \pi_{\boldsymbol{\theta}}(a_t|s_t)]}_{\text{NN Loss function}}};$$
  end
until forever ;

Policy Gradient Methods

Example REINFORCE-NN-Torch-CartPole.py REINFORCE with NN for Cart-Pole Environment (small network: $1 \times 32 \times 32 \times 2$). Compared to Semi-Gradient SARSA



Policy Gradient Methods

REINFORCE with baseline

- It is the basis for **Actor-Critic** methods, where two networks are used:
  - Critic. Estimates the value function ($\hat{v}_{\mathbf{w}}(s)$)
  - Actor. Policy based on critic estimations ($\hat{\pi}_{\boldsymbol{\theta}}(a|s)$)
- PGT

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \propto \mathbf{E}_{\pi}\left[\sum_a q_\pi(s,a) \nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a|s)\right]$$

can be generalized to compare $q_\pi(s,a)$ against anything not depending on a

$$\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}) \propto \mathbf{E}_{\pi}\left[\sum_a (q_\pi(s,a) - \underbrace{b(s)}_{\text{baseline}}) \nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a|s)\right]$$

because:

$$\sum_a b(s) \nabla_{\boldsymbol{\theta}} \pi_{\boldsymbol{\theta}}(a|s) = b(s) \nabla_{\boldsymbol{\theta}} \underbrace{\sum_a \pi_{\boldsymbol{\theta}}(a|s)}_{1} = b(s) \nabla_{\boldsymbol{\theta}} 1 = 0$$

- $b(s)$ can take any value (independent on a), i.e.
  - Average on a of $q_\pi(s,a)$
  - or more smart: $\hat{v}_\pi(s)$

Policy Gradient Methods

REINFORCE with baseline

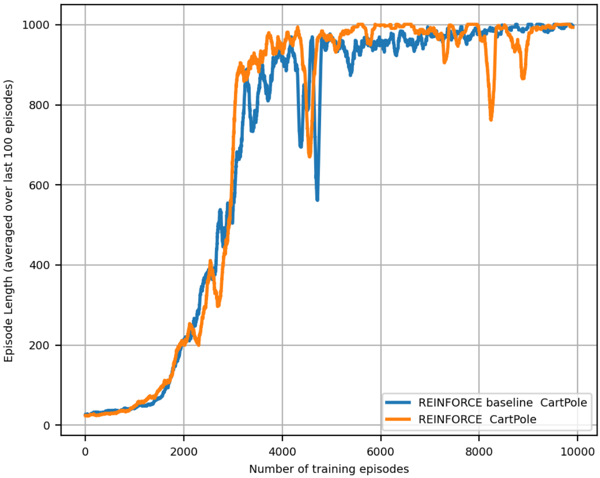
$\pi_{\boldsymbol{\theta}}(a|s)$, differentiable policy function ;
$\hat{v}_{\boldsymbol{w}}(s)$, differentiable state-value ;
$0 < \alpha_{\boldsymbol{w}}, \alpha_{\boldsymbol{\theta}} < 1$, learning rates ;
$\boldsymbol{\theta}, \boldsymbol{w} \leftarrow 0$;
repeat
  Generate episode: $S_0, A_0, R_1, S_1, \cdots A_{T-1}, R_T$, following $\pi_{\boldsymbol{\theta}}$;
  for $t \in 0 \cdots T-1$ **do**
    $\delta \leftarrow G_t - \hat{v}_{\boldsymbol{w}}(S_t)$; $\longrightarrow$ baseline
    $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha_{\boldsymbol{w}} \gamma^t \delta \boldsymbol{\nabla}_{\boldsymbol{w}} \hat{v}_{\boldsymbol{w}}(s_t)$; $\longrightarrow$ value estimator (critic)
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha_{\boldsymbol{\theta}} \gamma^t \delta \boldsymbol{\nabla}_{\boldsymbol{\theta}} \ln \pi_{\boldsymbol{\theta}}(a_t|s_t)$; $\longrightarrow$ policy estimator (actor)
  end
until forever ;

Note that:

$$\begin{aligned} \delta \boldsymbol{\nabla}_{\boldsymbol{w}} \hat{v}_{\boldsymbol{w}} &= G_t \boldsymbol{\nabla}_{\boldsymbol{w}} \hat{v}_{\boldsymbol{w}} - \hat{v}_{\boldsymbol{w}} \boldsymbol{\nabla}_{\boldsymbol{w}} \hat{v}_{\boldsymbol{w}} = G_t \boldsymbol{\nabla}_{\boldsymbol{w}} \hat{v}_{\boldsymbol{w}} - \frac{1}{2} \boldsymbol{\nabla}_{\boldsymbol{w}} \hat{v}_{\boldsymbol{w}}^2 \\ &= \boldsymbol{\nabla}_{\boldsymbol{w}} \left(G_t \hat{v}_{\boldsymbol{w}} - \frac{1}{2} \hat{v}_{\boldsymbol{w}}^2 \right) \longrightarrow \text{NN loss function} \end{aligned}$$

Policy Gradient Methods

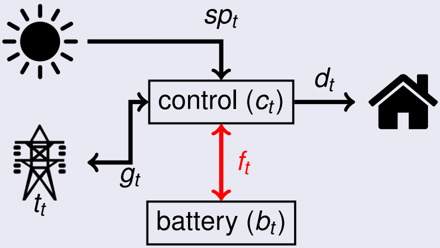
Example REINFORCE-baseline-NN-Torch-CartPole.py REINFORCE baseline with NN for Cart-Pole Environment (2 small networks: $1 \times 32 \times 32 \times 2$). Compared REINFORCE



Contents

Home Solar Example Control

Home Solar



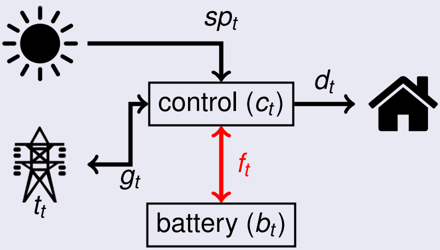
- sp_t: solar power at slot t
- d_t: home energy demand
- g_t: energy to/from the grid
- $t_t(sign(g_t))$: tariff (energy price, buy or sell)
- b_t: battery energy
- c_t: control action
- f_t: battery flow

Home Solar constraints

- Optimize one day episodes. Time slot 30'. Episode: 48 slots
- Fixed demand. $[d_0, \ldots d_{47}]$, $d_t \in \mathbb{R}_+$
- Solar power obtained from several months
- Battery capacity fixed (B). $[b_0, \ldots b_{47}]$, $0 \leq b_t \leq B$

Home Solar Example Control

Home Solar



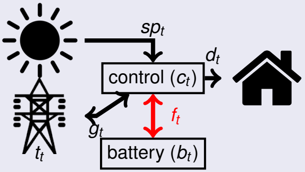
- sp_t: solar power at slot t
- d_t: home energy demand
- g_t: energy to/from the grid
- $t_t(sign(g_t))$: tariff (energy price, buy or sell)
- b_t: battery energy
- c_t: control action
- f_t: battery flow

Home Solar constraints

- Discretized charge/discharge battery (N levels, p: step power)
- $f_t \in \{0, p, 2p, \ldots (N-1)p, -p, -2p, \ldots -(N-1)p\}$
- Control determines battery action: $c_t \in \{0, 1, 2, \ldots (N-1), -1, -2, \ldots, -(N-1)\}$
- $|\mathcal{A}| = 1 + 2(N-1)$

Home Solar Example Control

Home Solar environment



state:
$s_t = [t, b_t],\ t \in \{0 \ldots 47\},\ b_t \in \mathbb{R}_+,\ \text{s.t. } 0 \le b_t \le B$
action (one hot encoded):
$a_t : [a_{1,t}, \ldots a_{|\mathcal{A}|,t}],\ a_{\cdot,t} \in \{0, 1\} \text{ s.t. } \sum_{i=1}^{|\mathcal{A}|} a_{i,t} = 1$
$charge\_levels = [0, p, 2p, \ldots (N-1)p, -p, \ldots -(N-1)p]$
$charge_t = \sum charge\_levels \odot a_t$

element wise product (Hadamard)

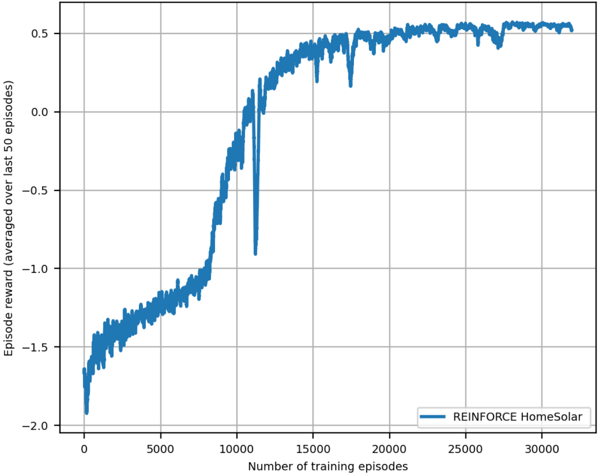
case $charge_t == 0$ **do**
| $f_t \leftarrow 0$
case $charge_t > 0$ **do**
| $f_t \leftarrow \min(charge_t/2, B - b_t)$ → charging battery
case $charge_t < 0$ **do**
| $f_t \leftarrow \max(charge_t/2, -b_t)$ → discharging battery
end

30' slot

$b_{t+1} = b_t + f_t$
$g_t = d_t - sp_t + f_t$
$reward_t = -g_t \cdot t_t$

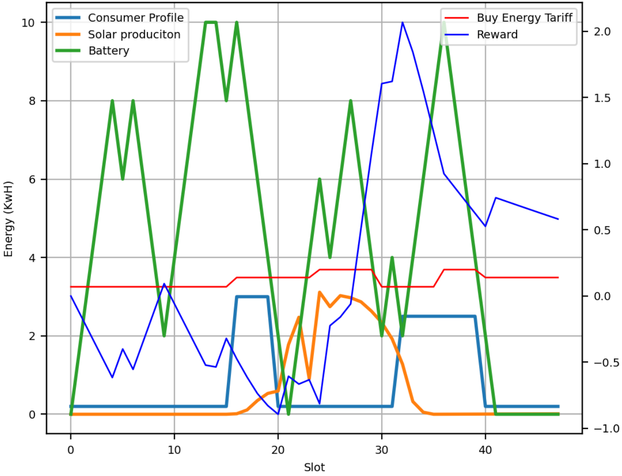
Home Solar Example Control

Example `home-project/main.py` REINFORCE trained with one day



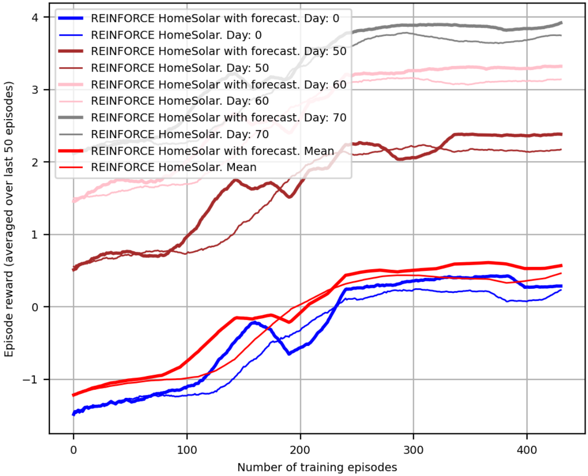
Home Solar Example Control

Example `home-project/main.py` REINFORCE trained with one day



Home Solar Example Control

Example `home-project/main2.py` REINFORCE trained with all dataset with solar power forecast



Contents

Bibliography

- Reinforcement Learning: An Introduction. Richard S. Sutton and Andrew G. Barto. MIT Press, Cambridge, MA, 2018