

# Business 4720 - Class 21

## Reinforcement Learning – Tabular Methods

Joerg Evermann

Faculty of Business Administration  
Memorial University of Newfoundland  
`jevermann@mun.ca`



Unless otherwise indicated, the copyright in this material is owned by Joerg Evermann. This material is licensed to you under the [Creative Commons by-attribution non-commercial license \(CC BY-NC 4.0\)](#)

## What You Will Learn:

- ▶ Reinforcement Learning
  - ▶ Multi-Armed Bandits
  - ▶ Value Functions
  - ▶ Monte-Carlo (MC) Methods
  - ▶ Temporal-Difference (TD) Methods

Richard S. Sutton and Andrew G. Barto (2018) *Reinforcement Learning – An Introduction*. 2nd edition, The MIT Press, Cambridge, MA. (SB)

<http://incompleteideas.net/book/the-book.html>

Chapters 2–7

(CC BY-NC-ND License)

Implementations are available on the following GitHub repo:

`https://github.com/jevermann/busi4720-rl`

The project can be cloned from this URL:

`https://github.com/jevermann/busi4720-rl.git`

## Online Learning by Acting in an Environment

### Ideas

- ▶ Maximize return
- ▶ Immediate and delayed/subsequent rewards
- ▶ Discover which actions to take by trying them
- ▶ Tradeoff between exploration and exploitation
- ▶ Uncertain/random/stochastic environment

*Problem cannot be tackled by optimization (e.g. dynamic programming), because of incomplete knowledge of environment.*

## Core Elements

- ▶ **Policy**  $\pi$  (probability of taking action  $a$  in state  $s$ )
- ▶ **Reward**  $R$  (received from the environment after each action)
- ▶ **Return**  $G$  (possibly discounted sum of future rewards)
- ▶ **State value function**  $v$  (expected return for each state)
- ▶ **Action value function**  $q$   
(expected return for each state and action)
- ▶ **Model**  $p$  (behaviour of the environment, optional)

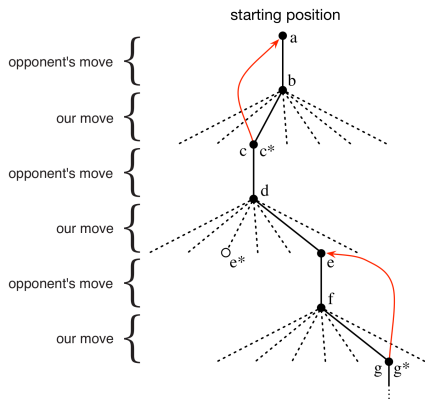
# Introductory Example – Tic-Tac-Toe (Naughts and Crosses)



[https://commons.wikimedia.org/wiki/File:Tic\\_tac\\_toe.svg](https://commons.wikimedia.org/wiki/File:Tic_tac_toe.svg)

- ▶ Value is probability of winning from state
- ▶ Value of any state with X-X-X is 1.0
- ▶ Value of any state with O-O-O or full board is 0.0
- ▶ Initial values are 0.5

# Introductory Example – Tic-Tac-Toe [cont'd]



Source: SB Figure 1.1

- ▶ Normally, move *greedily* to highest-valued next state
- ▶ Sometimes, make random *exploratory* move
- ▶ After each greedy move, update state value towards value of later state with *step size*  $\alpha$
- ▶ *Temporal-difference* learning  $V(S_{t+1}) - V(S_t)$
- ▶ Take advantage of information *during* episode/game.

$$V(S_t) \leftarrow V(S_t) + \alpha [V(S_{t+1}) - V(S_t)]$$



# Example Applications in Business and Management

- ▶ **Marketing:** Learn which online ads to show to which site visitor
- ▶ **HR:** Learn which employee to assign to which task
- ▶ **Operations:** Learn which work item to assign to which machine
- ▶ **Logistics:** Learn which item to route on which truck or flight
- ▶ ...

# K-armed Bandits



[https://commons.wikimedia.org/wiki/File:Antique\\_one-armed\\_bandit,\\_Ventnor,\\_Isle\\_of\\_Wight,\\_UK.jpg](https://commons.wikimedia.org/wiki/File:Antique_one-armed_bandit,_Ventnor,_Isle_of_Wight,_UK.jpg)

- ▶ Stateless environment
- ▶  $k$  possible actions  $A_t$  at time  $t$  with stochastic reward  $R_t$

- ▶ Estimate *action value* as:

$$Q_t(a) = \frac{\sum_{i=1}^{t-1} R_i \times \mathbb{1}_a}{\sum_{i=1}^{t-1} \mathbb{1}_a} \quad (\text{average reward})$$

- ▶  $\epsilon$ -greedy policy: With probability  $\epsilon$  take random action, with probability  $1 - \epsilon$  take optimal action

$$A_t = \underset{a}{\operatorname{argmax}} Q_t(a)$$

- ▶ Incremental implementation

$$Q_{t+1}(a) = Q_t(a) + \frac{1}{t} [R_t(a) - Q_t(a)]$$

# General Update Rule for Estimates

$$NewEstimate \leftarrow OldEstimate + StepSize [Target - OldEstimate]$$

$[Target - OldEstimate]$  is the *error* in the estimate

# K-armed Bandits – Example

## A simple bandit algorithm

Initialize, for  $a = 1$  to  $k$ :

$$Q(a) \leftarrow 0$$

$$N(a) \leftarrow 0$$

Loop forever:

$$A \leftarrow \begin{cases} \operatorname{argmax}_a Q(a) & \text{with probability } 1 - \epsilon \\ \text{a random action} & \text{with probability } \epsilon \end{cases}$$

$$R \leftarrow \text{bandit}(A)$$

$$N(A) \leftarrow N(A) + 1$$

$$Q(A) \leftarrow Q(A) + \frac{1}{N(A)} [R - Q(A)]$$

# K-armed Bandits – Python

Environment:

```
class k_bandit_env:
    def __init__(self, k):
        self.k = k
        self.mean_rewards = []

        for i in range(self.k):
            self.mean_rewards \
                .append(random.normalvariate(0, 1))

    def step(self, action):
        mean = self.mean_rewards[action]
        reward = random.normalvariate(mean, 1)
        return reward
```

# K-armed Bandits – Python

Agent:

```
class k_bandit_agent:
    def __init__(self, k, epsilon, initial_value):
        self.k = k
        self.epsilon = epsilon
        self.env = k_bandit_env(k)

        self.Q = [initial_value] * self.k
        self.N = [.0] * self.k

    def determine_action(self):
        if random.uniform(0,1) < self.epsilon:
            # explore
            action = random.randint(0, self.k-1)
        else:
            # exploit
            action = self.Q.index(max(self.Q))
        return action
```

## Agent (continued):

```
def train(self, steps):  
    rewards = []  
    for step in range(steps):  
        action = self.determine_action()  
        reward = self.env.step(action)  
        self.N[action] += 1  
        self.Q[action] = \  
            (reward - self.Q[action]) / self.N[action]  
        rewards.append(reward)  
    return rewards
```

Complete implementation at

<https://evermann.ca/busi4720/bandits.py>

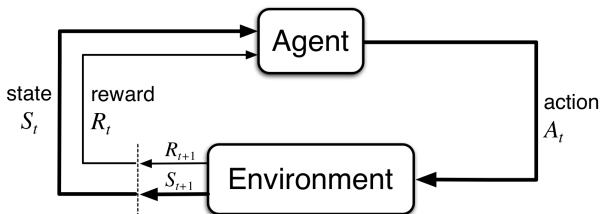
# K-armed Bandits – Policy Comparison

Values for  $\epsilon$  and the initial  $Q$  value affect learning behaviour:





# Markov Decision Processes



Source: SB Figure 3.1

## ► Trajectory:

$$S_0, A_0, R_1, S_1, A_1, R_2, \dots$$

## ► Dynamics:

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

- **Discounted future return:**

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned}$$

- **State value function** of state  $s$  under a policy  $\pi$ :

$$v_{\pi}(s) = \mathbb{E}_{\pi}[G_t | S_t = s] = \mathbb{E}_{\pi} \left[ \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s \right]$$

- **Action value function** for policy  $\pi$ :

$$\begin{aligned} q_{\pi}(s, a) &= \mathbb{E}_{\pi}[G_t | S_t = s, A_t = a] \\ &= \mathbb{E}_{\pi} \left[ \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a \right] \end{aligned}$$

# Bellman Equation

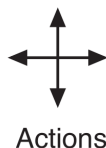
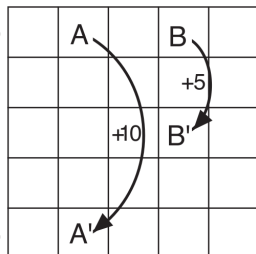
The value function  $v$  for policy  $\pi$  is the unique solution to the **Bellman equation**:

$$\begin{aligned}v_{\pi}(s) &= \mathbb{E}_{\pi}[G_t | S_t = s] \\&= \mathbb{E}_{\pi}[R_{t+1} + \gamma G_{t+1} | S_t = s] \\&= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r | s, a) [r + \gamma \mathbb{E}_{\pi}[G_{t+1} | S_{t+1} = s']] \\&= \sum_a \pi(a|s) \sum_{s', r} p(s', r | s, a) [r + \gamma v_{\pi}(s')] \quad \text{for all } s \in \mathcal{S}\end{aligned}$$

(Recall: Expected value is sum weighted by probabilities)

# MDP – Gridworld State Value Function

- ▶ Actions: Up, Down, Left, Right
- ▶ Falling off the world results in reward of  $-1$
- ▶ Other actions result in reward of  $0$ , except for  $A$  to  $A'$  and  $B$  to  $B'$  as indicated
- ▶ Policy  $\pi$  is to take each action with equal probability
- ▶ Discount rate  $\gamma = 0.9$



3.3	8.8	4.4	5.3	1.5
1.5	3.0	2.3	1.9	0.5
0.1	0.7	0.7	0.4	-0.4
-1.0	-0.4	-0.4	-0.6	-1.2
-1.9	-1.3	-1.2	-1.4	-2.0

Source: SB Figure 3.2

# MDP – Optimal Policies

Maximizing the state value function  $v$  or action value function  $q$  is finding an optimal policy  $\pi$ :

$$v_*(s) = \max_{\pi} v_{\pi}(s)$$

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

Intuitively, the value of a state under an optimal policy  $\pi_*$  is equal to the expected return for the the best action from that state:

$$v_*(s) = \max_{a \in \mathcal{A}(s)} q_{\pi_*}(s, a)$$

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

Similarly for action value under an optimal policy:

$$\begin{aligned}q_*(s, a) &= \mathbb{E} \left[ R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') | S_t = s, A_t = a \right] \\&= \sum_{s', r} p(s', r | s, a)[r + \gamma \max_{a'} q_*(s', a')]\end{aligned}$$

## Intuition:

- 1 Start from random value function and policy
- 2 Compute updated value function (iteratively)
- 3 Adjust policy based on updated value function
- 4 Repeat from (2) until convergence

## Iterative Policy Evaluation

Loop:

$$\Delta \leftarrow 0$$

Loop for each  $s \in \mathcal{S}$  :

$$v \leftarrow V(s)$$

$$V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)[r + \gamma V(s')]$$

$$\Delta \leftarrow \max(\Delta, |v - V(s)|)$$

until  $\Delta < \theta$



# Iterative Policy Evaluation in Python

```
# Initialize value function V
V = dict()
for state in States:
    V[state] = 0
# Initialize policy pi
pi = dict()
for state in States:
    pi[state] = 0

def evaluate_policy():
    while True:
        Delta = 0
        for s in States:
            v = V[s]
            V[s] = exp_reward(s, pi[s])
            Delta = max(Delta, abs(v - V[s]))
        print(Delta)
        if Delta < theta:
            break
```

## Iterative Policy Improvement

Loop:

*stable*  $\leftarrow$  *true*

For each  $s \in \mathcal{S}$  :

*old\_action*  $\leftarrow \pi(s)$

$\pi(s) \leftarrow \operatorname{argmax}_a \sum_{s', r} p(s', r | s, a) [r + \gamma V(s')]$

If *old\_action*  $\neq \pi(s)$  then *stable*  $\leftarrow$  *false*

If *stable* then

return  $V \approx v_*$  and  $\pi \approx \pi_*$

else

go to policy evaluation

# Iterative Policy Improvement in Python

```
def improve_policy():  
    stable = True  
    for s in States:  
        old_action = pi[s]  
        max_r = -math.inf  
        max_a = None  
        for action in Actions:  
            r = exp_reward(s, action)  
            if r > max_r:  
                max_r = r  
                max_a = action  
        pi[s] = max_a  
        if old_action != pi[s]:  
            stable = False  
    return stable
```

# Iterative Policy Improvement in Python

```
stable = False
while not stable:
    evaluate_policy()
    stable = improve_policy()

print("Optimal Policy:")
print(pi)
```

Complete implementation at

<https://evermann.ca/busi4720/jacks.py>

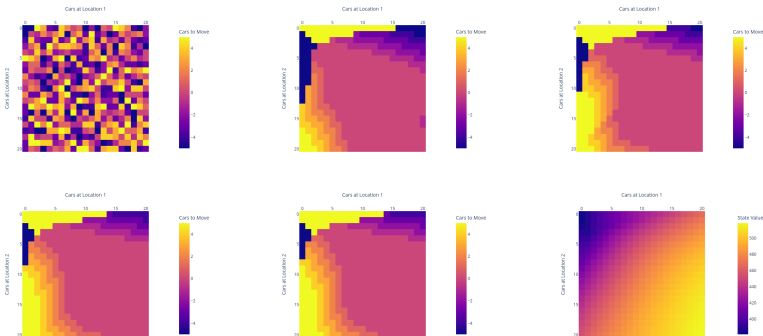
# Example – Jack's Car Rental

- ▶ Jack rents cars at 2 locations with capacity of 20 cars
- ▶ Daily rental requests and returns are Poisson distributed
- ▶ Move a maximum of 5 cars between locations overnight
- ▶  $-2$  reward for each move,  $+10$  reward for each satisfied rental request
- ▶ How many vehicles to move each night?

```
States = []  
for cars1 in range(21):  
    for cars2 in range(21):  
        States.append((cars1, cars2))  
  
Actions = range(-5, 5+1)
```

# Example – Policies and Final Value Function

Policies after each improvement, starting with random initial policy. Final state value function at bottom right.



# Monte Carlo Methods

- ▶ In practice, dynamics of the environment are not known, i.e.  $p(s', r|s, a)$  is unknown; there is *no model* of the environment
- ▶ Learning  $V$  and  $Q$  from *experience*, ie. sample sequences of states, actions, and rewards.
- ▶ Consider *episodic tasks*, with a terminal state and finite returns
- ▶ Create episodes following policy  $\pi$  by interacting with environment
- ▶ Similar to the bandit problem, which also learned an action value function, but now with states

# First-visit MC Prediction

- ▶ Estimate  $V \approx v_\pi$
- ▶ Return assigned to state is that of first visit of state

Input: a policy  $\pi$  to be evaluated

Initialize:

$V(s) \in \mathbb{R}$ , arbitrarily, for all  $s \in \mathcal{S}$

$Returns(s) \leftarrow$  an empty list, for all  $s \in \mathcal{S}$

Loop forever (for each episode):

Generate an episode following  $\pi : S_0, A_0, R_1, S_1, A_1, R_2, \dots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

Loop for each step of episode,  $t = T - 1, T - 2, \dots, 0$  :

$G \leftarrow \gamma G + R_{t+1}$

Unless  $S_t$  appears in  $S_0, S_1, \dots, S_{t-1}$  :

Append  $G$  to  $Returns(S_t)$

$V(S_t) \leftarrow \text{average}(Returns(S_t))$



- ▶ State-value function  $V$  not useful without model
- ▶ Estimate action-value function  $Q$  and  $\pi \approx \pi_*$  directly
- ▶ To ensure all state-action pairs are visited with a greedy policy, set these as episode starts with some probability (*"exploring starts", ES*)

# MC Control (ES)

Initialize for all  $s \in \mathcal{S}, a \in \mathcal{A}$

$\pi(s) \in \mathcal{A}(s)$  (arbitrarily)

$Q(s, a) \in \mathbb{R}$  (arbitrarily)

$Returns(s, a) \leftarrow$  empty list

Loop forever (for each episode):

Choose  $S_0 \in \mathcal{S}, A_0 \in \mathcal{A}(S_0)$  randomly

Generate an episode following  $\pi : S_0, A_0, R_1, S_1, A_1, R_2, \dots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

Loop for each step of episode,  $t = T - 1, T - 2, \dots, 0$  :

$G \leftarrow \gamma G + R_{t+1}$

Unless the pair  $S_t, A_t$  appears in  $S_0, A_0, S_1, A_1, \dots, S_{t-1}, A_{t-1}$  :

Append  $G$  to  $Returns(S_t, A_t)$

$Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$

$\pi(S_t) \leftarrow \operatorname{argmax}_a Q(S_t, a)$

# MC (ES) Control Example – Black Jack

- ▶ Card values A, 2, 3, ..., 10
- ▶ Ace can be 1 or 11 ("usable ace")
- ▶ Actions: take another card ("hit") or do not ("stick")
- ▶ Dealer showing initial card, stands on 17 or more
- ▶ Over 21 is "bust" (lost), otherwise closest to 21 wins

# MC (ES) Control Example – Black Jack

Define states  $\mathcal{S}$  and actions  $\mathcal{A}$ :

```
gamma = 1.0
# Define states
States = []
for ace in [0,1]:
    for dealer_showing in range(1,11):
        for hand_sum in range(12, 22):
            States.append( (ace,dealer_showing,hand_sum) )

# Define actions
Actions = (0, 1)
```

# MC (ES) Control Example – Black Jack

Initialize policy  $\pi$ , action-value function  $Q$ , and returns:

```
# Initialize policy
pi = dict()
for s in States:
    pi[s] = random.randint(0,1)

# Initialize action value function
Q = dict()
for s in States:
    for a in Actions:
        Q[(s, a)] = 0

# Initialize returns
Returns = dict()
for s in States:
    for a in Actions:
        Returns[(s, a)] = []
```

# MC (ES) Control Example – Black Jack

Generate an episode using policy  $\pi$  from initial state  $S_0$  and initial action  $A_0$ :

```
def generate_episode(pi, s0, a0):
    terminal = False
    s = s0
    a = a0
    states = [s0]
    actions = [a0]
    rewards = [math.nan]
    while terminal is False:
        sprime, r, terminal = step(s, a)
        rewards.append(r)
        if not terminal:
            aprime = pi[sprime]
            states.append(sprime)
            actions.append(aprime)
            s = sprime
            a = aprime

    return states, actions, rewards, len(rewards)
```

# MC (ES) Control Example – Black Jack

Learn the  $Q$  function:

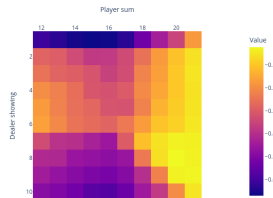
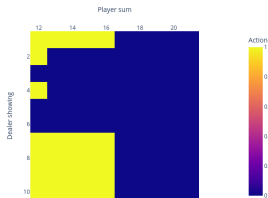
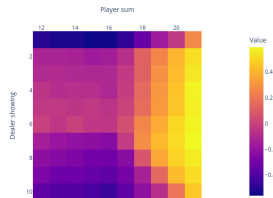
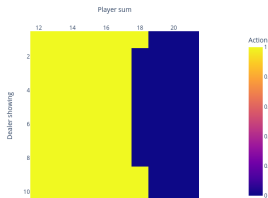
```
for e in range(0, 1000000+1):
    s0 = random.choice(States)
    pi0 = random.choice(Actions)
    S, A, R, T = generate_episode(pi, s0, pi0)
    G = 0
    for t in reversed(range(0, T-1)):
        G = gamma*G + R[t+1]
        if (t == 0) or \
            ((S[t], A[t]) \
             not in zip(S[0:t-1], A[0:t-1])):
            Returns[(S[t], A[t])].append(G)
            Q[(S[t], A[t])] = mean>Returns[(S[t], A[t])])
            if Q[(S[t], 1)] > Q[(S[t], 0)]:
                pi[S[t]] = 1
            else:
                pi[S[t]] = 0
```

Full example available at

[https://evermann.ca/bus14720/blackjack\\_es.py](https://evermann.ca/bus14720/blackjack_es.py)

# MC (ES) Control Example – Black Jack

Policies and state value function after 1,000,000 episodes.  
Usable ace on top, no usable ace at bottom:





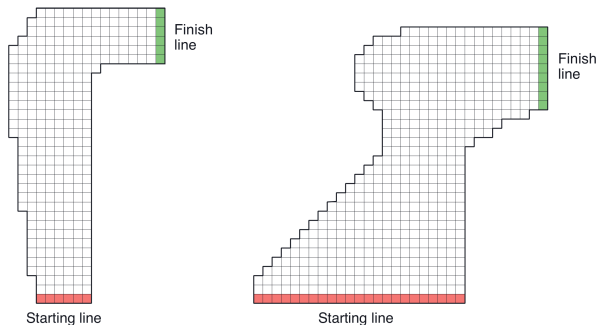
# MC Control Example – Epsilon-Soft Policy

- ▶ Exploring starts not always feasible
- ▶ Use  $\epsilon$ -soft policy to ensure exploration (instead of greedy policy)
- ▶ Policy  $\pi$  is now the probability of taking action  $a$  in state  $s$
- ▶ Update  $\pi$  with epsilon-soft probabilities, instead of greedy deterministic action

```
Q[(S[t], A[t])] = mean>Returns[(S[t], A[t])])
# Optimal policy (for two actions)
A_star = 1 if Q[(S[t], 1)] > Q[(S[t], 0)] else 0

for a in Actions:
    if a == A_star:
        pi[(S[t], a)] = 1-epsilon+epsilon/len(Actions)
    else:
        pi[(S[t], a)] = epsilon/len(Actions)
```

# MC Control Example – Racetrack



Source: SB Figure 5.5

- **States:** Position and velocity on race track
- **Actions:** Accelerate +1, 0, -1 in horizontal or vertical dir.
- **Rewards:** -1 for each step, +1 for reaching finish line
- **Noise:** With small probability, actions are ignored

# MC Control Example – Racetrack

```
Actions = []
for y in range(-1, 2):
    for x in range(-1, 2):
        Actions.append((y,x))

Q = dict()
def getQ(s, a):
    if (s, a) not in Q:
        return 0
    else:
        return Q[(s, a)]

pi = dict()
def get_action(s):
    weights = []
    for a in Actions:
        if (s, a) in pi:
            weights.append(pi[(s, a)])
    return random.choices(Actions, weights=weights)[0]
```

# MC Control Example – Racetrack

```
Returns = dict()
def getReturns(s, a):
    if (s, a) not in Returns:
        return []
    else:
        return Returns[(s, a)]
def appendReturn(s, a, r):
    if (s, a) not in Returns:
        Returns[(s, a)] = [r]
    else:
        Returns[(s, a)].append(r)
```

# MC Control Example – Racetrack

```
for e in range(0, 10000+1):
    S, A, R, T = env.generate_episode()
    G = 0
    for t in reversed(range(0, T-1)):
        G = gamma*G + R[t+1]
        if (t == 0) or ((S[t], A[t]) \
            not in zip(S[0:t-1], A[0:t-1])):
            appendReturn(S[t], A[t], G)
            Q[(S[t],A[t])] = mean(getReturns(S[t],A[t]))
            A_star = argmaxQ(S[t])
            for a in Actions:
                if a == A_star:
                    pi[(S[t],a)] = 1-eps+eps/len(Actions)
                else:
                    pi[(S[t],a)] = eps/len(Actions)
```

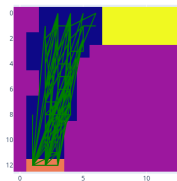
Full example at

<https://evermann.ca/busi4720/racetrack.py>

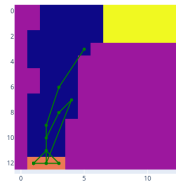
# MC Control Example – Racetrack

Racetrack trajectory after 0, 100, 200, and 10000 learning episodes:

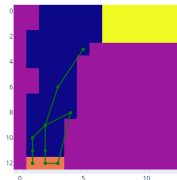
T=4953



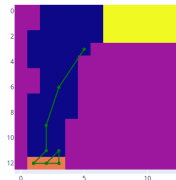
T=19



T=16



T=12



# On-Policy and Off-Policy Learning

## On-Policy Learning

- ▶ Trying to learn action values for optimal behaviour
- ▶ But have to behave sub-optimal (e.g.  $\epsilon$ -soft) to explore all actions

## Off-Policy Learning

- ▶ Use two policies
- ▶ **Target policy**  $\pi$ : Being learned, typically deterministic, greedy
- ▶ **Behaviour policy**  $b$ : Used to generate behaviour (episodes), typically  $\epsilon$ -soft
- ▶ Behaviour policy  $b$  must **cover** target policy  $\pi$  (i.e. all possible behaviour under  $\pi$  must be generated by  $b$ )

# Off-Policy MC Control

Initialize for all  $s \in \mathcal{S}, a \in \mathcal{A}(s)$  :

$Q(s, a) \in \mathbb{R}$  (arbitrarily)

$C(s, a) \leftarrow 0$

$\pi(s) \leftarrow \operatorname{argmax}_a Q(s, a)$

Loop forever (for each episode):

Generate an episode following  $b : S_0, A_0, R_1, S_1, A_1, R_2, \dots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0; W \leftarrow 1$

Loop for each step of episode,  $t = T - 1, T - 2, \dots, 0$  :

$G \leftarrow \gamma G + R_{t+1}$

$C(S_t, A_t) \leftarrow C(S_t, A_t) + W$

$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)} [G - Q(S_t, A_t)]$

$\pi(S_t) \leftarrow \operatorname{argmax}_a Q(S_t, a)$

If  $A_t \neq \pi(S_t)$  then proceed to next episode

$W \leftarrow W/b(A_t|S_t)$



# Off-Policy MC Control in Python – Policies

## Racetrack example revisited

```
def b(s):
    weights = []
    for a in Actions:
        if (s, a) in Q:
            weights.append(math.exp(Q[(s, a)]))
        else:
            weights.append(0)
    if len(weights) == 0 or sum(weights) == 0:
        return random.choice(Actions)
    else:
        return random.choices(Actions, weights)[0]

def pi(s):
    a = argmaxQ(s)
    if a is None:
        return random.choice(Actions)
    else:
        return a
```

# Off-Policy MC Control in Python – Policies

Racetrack example revisited

```
def bprob(a, s):
    if (s, a) not in Q:
        return 1
    weights = []
    for aa in Actions:
        if (s, aa) in Q:
            weights.append(math.exp(Q[(s, aa)]))
    if len(weights) == 0 or sum(weights) == 0:
        return 1
    else:
        return math.exp(Q[(s, a)]) / sum(weights)
```

# Off-Policy MC Control in Python – Learning

## Racetrack example revisited

```
for e in range(0, 10000+1):
    S, A, R, T = env.generate_episode_b()
    G = 0
    W = 1
    for t in reversed(range(0, T-1)):
        G = gamma*G + R[t+1]
        C[(S[t], A[t])] = getC(S[t], A[t]) + W
        Q[(S[t], A[t])] = getQ(S[t], A[t]) + \
            W/getC(S[t], A[t]) * (G-getQ(S[t],A[t]))
        if A[t] != pi(S[t]):
            break
    else:
        W = W * 1/bprob(A[t], S[t])
```

Full example at

[https://evermann.ca/busi4720/racetrack\\_off\\_policy.py](https://evermann.ca/busi4720/racetrack_off_policy.py)

# Temporal-Difference Learning

## MC Control

- ▶ Waits until end of episode before updating  $Q$
- ▶ Updates based on target (discounted) return  $G_t$

$$Q(S_t, a) \leftarrow Q(S_t, a) + \alpha [G_t - Q(S_t, a)]$$

## TD Control

- ▶ Why wait?
- ▶ Updates based on target of reward plus discounted future expected return under the optimal action:

$$Q(S_t, a) \leftarrow Q(S_t, a) + \alpha [R_{t+1} + \gamma Q(S_{t+1}, a_{t+1}^*) - Q(S_t, a)]$$

# On-Policy TD Learning – SARSA

Initialize  $Q(s, a)$  for all  $s \in \mathcal{S}^+$ , arbitrarily

Loop for each episode:

    Initialize  $S$

    Choose  $A$  from  $S$  using policy derived from  $Q$

    Loop for each step of episode:

        Take action  $A$ , observe  $R, S'$

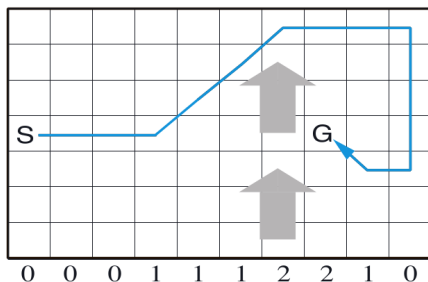
        Choose  $A'$  from  $S'$  using policy derived from  $Q$

$$Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma Q(S', A') - Q(S, A)]$$

$$S \leftarrow S'; A \leftarrow A'$$

    until  $S$  is terminal

# SARSA Example – Windyworld



Source: SB Chapter 6

- ▶ Non-discounted ( $\gamma = 1$ )
- ▶ Rewards are -1 until termination
- ▶ No penalties for moving off-world

# SARSA Example – Windyworld

```
# Define states
States = []
for i in range(nrow):
    for j in range(ncol):
        States.append((i, j))
# Define actions
Actions = range(0, 4)
# Initialize Q
Q = dict()
for s in States:
    for a in Actions:
        Q[(s, a)] = random.random()
# Define pi
def pi(s):
    if random.random() < epsilon:
        return random.choice(Actions)
    else:
        return argmaxQ(s)
```

# SARSA Example – Windyworld

```
for e in range(0, 100):
    terminal = False
    S = windy.reset()
    A = pi(S)
    step = 0
    while terminal is False:
        Sprime, R, terminal = windy.step(A)
        Aprime = pi(Sprime)
        Q[(S,A)] = Q[(S,A)] + alpha*(R + \
            gamma * Q[(Sprime, Aprime)] - Q[(S, A)])
        S = Sprime
        A = Aprime
```

Complete example at [https:](https://evermann.ca/busi4720/windyworld_sarsa.py)

[//evermann.ca/busi4720/windyworld\\_sarsa.py](https://evermann.ca/busi4720/windyworld_sarsa.py)



# Generalizing SARSA to n-Step TD Control

## TD Error (1-Step Error):

$$\delta_{TD} = R_{t+1} + Q(S_{t+1}, a_{t+1}) - Q(S_t, a)$$

Recall that:

$$Q(S_t, A_t) = \mathbb{E}[G_t | S_t, A_t] \quad \text{and} \quad G_t = R_{t+1} + \gamma G_{t+1}$$

## TD Error (2-Step Error):

$$\delta_{TD} = R_{t+1} + \gamma R_{t+2} + \gamma^2 Q(S_{t+2}, a_{t+2}) - Q(S_t, a)$$

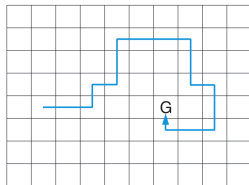
...

## TD Error (n-Step Error):

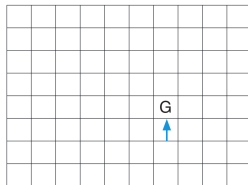
$$\delta_{TD} = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n Q(S_{t+n}, a_{t+1}) - Q(S_t, a)$$

# Generalizing SARSA to n-Step TD Control

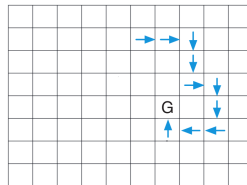
Path taken



Action values increased  
by one-step Sarsa



Action values increased  
by 10-step Sarsa



Source: SB Figure 7.4

# Off-Policy TD Learning – Q-Learning

Initialize  $Q(s, a)$  for all  $s \in \mathcal{S}^+$ , arbitrarily

Loop for each episode:

Initialize  $S$

Loop for each step of episode:

Choose  $A$  from  $S$  using policy derived from  $Q$

Take action  $A$ , observe  $R, S'$

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

$$S \leftarrow S'$$

until  $S$  is terminal

# Q-Learning Example – Windyworld

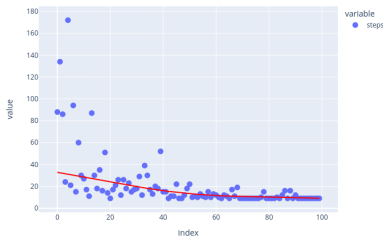
```
for e in range(0, 1000):  
    terminal = False  
    S = windy.reset()  
    step = 0  
    while terminal is False:  
        A = pi(S)  
        Sprime, R, terminal = windy.step(A)  
        Q[(S,A)] = Q[(S,A)] + alpha*(R + \  
            gamma * maxQ(Sprime) - Q[(S, A)])  
        S = Sprime
```

Complete example at [https://evermann.ca/busi4720/windyworld\\_q\\_learning.py](https://evermann.ca/busi4720/windyworld_q_learning.py)

# SARSA and Q-Learning Results on Windyworld

Steps per episode to termination:

## SARSA



## Q-Learning

