Lecture F4. MDP without Model - Policy Iteration (Q-learning, double Q-learning)

Sim, Min Kyu, Ph.D., mksim@seoultech.ac.kr

SEOULTECH 서울과학기술대학교 데이터사이언스학과

- `skiier.R` is loaded as follows.

```
source("../skiier.R")
```

```
## [1] "Skiier's problem is set."
## [1] "Defined are `state`, `P_normal`, `P_speed`, `R_s_a`, `q_s_a_init` (F2, p15)."
## [1] "Defined are `pi_speed`, and `pi_50` (F2, p16)."
## [1] "Defined are `simul_path()` (F2, p17)."
## [1] "Defined are `simul_step()` (F2, p18)."
## [1] "Defined are `pol_eval_MC()` (F2, p19)."
## [1] "Defined are `pol_eval_TD()` (F2, p20)."
## [1] "Defined are `pol_imp()` (F2, p20)."
```

I. Policy Iteration 3 - Q-learning control

Strategy

- (`pol_eval_MC()`) MC control updates $q(s, a)$:

$$q(s, a) \leftarrow q(s, a) + \alpha(G_t - q(s, a)), \ \forall s, a$$

- (`pol_eval_TD()`) TD control updates $q(s, a)$:

$$q(s, a) \leftarrow q(s, a) + \alpha(r_t + \gamma q(s', a') - q(s, a)), \ \forall s, a$$

- (`pol_eval_Q()`) Q-learning updates $q(s, a)$:

$$q(s, a) \leftarrow q(s, a) + \alpha(r_t + \gamma max_{a' \in \mathcal{A}} q(s', a') - q(s, a)), \ \forall s, a$$

- Q-learning is a variation of TD control.
- Q-learning is greedy in a sense by taking maximum among possible future action.
- It is called **off-policy learning** since it may take action other than the current policy dictates.

Write pol_eval_Q()

```
pol_eval_TD <- function(sample_step, q_s_a, alpha) {
  s <- sample_step[1]
  a <- sample_step[2]
  r <- sample_step[3] %>% as.numeric()
  s_next <- sample_step[4]
  a_next <- sample_step[5]
  q_s_a[s,a] <- q_s_a[s,a] + alpha*(r+q_s_a[s_next,a_next]-q_s_a[s,a])
  return(q_s_a)
}
```

```
pol_eval_Q <- function(sample_step, q_s_a, alpha) {
  s <- sample_step[1]
  a <- sample_step[2]
  r <- sample_step[3] %>% as.numeric()
  s_next <- sample_step[4]
  a_next <- sample_step[5] # not used here
  q_s_a[s,a] <- q_s_a[s,a] + alpha*(r+max(q_s_a[s_next,])-q_s_a[s,a]) # change here
  return(q_s_a)
}
```

Q-learning

```r
num_ep <- 10^5
beg_time <- Sys.time()
q_s_a <- q_s_a_init
pi <- pi_50
exploration_rate <- 1
for (epi_i in 1:num_ep) {
  s_now <- "0"
  while (s_now != "70") {
    sample_step <- simul_step(pi, s_now, P_normal, P_speed, R_s_a)
    q_s_a <- pol_eval_Q(sample_step, q_s_a, alpha = max(1/epi_i, 0.05))
    if (epi_i %% 100 == 0) {
      pi <- pol_imp(pi, q_s_a, epsilon = exploration_rate)
    }
    s_now <- sample_step[4]
    exploration_rate <- max(exploration_rate*0.9995, 0.001)
  }
}
end_time <- Sys.time()
```

```r
print(end_time-beg_time)
```

```
## Time difference of 10.06 secs
```

```r
t(pi)
```

```
##   0 10 20 30 40 50 60 70
## n 0  0  0  1  0  1  1  1
## s 1  1  1  0  1  0  0  0
```

```r
t(q_s_a)
```

```
##        0     10     20     30     40     50     60 70
## n -5.588 -4.662 -3.961 -2.695 -1.999 -2.000 -1.000  0
## s -5.017 -4.469 -3.362 -3.085 -1.673 -2.026 -1.685  0
```

II. Policy iteration 4 - Double Q-learning control

Method

- (`pol_eval_Q()`) Q-learning updates $q(s, a)$:

$$q(s,a) \leftarrow q(s,a) + \alpha(r_t + \gamma max_{a' \in \mathcal{A}} q(s', a') - q(s,a)), \ \ \forall s, a$$

- (`pol_eval_dbl_Q()`) Double Q-learning uses a pair of q-functions, $q_1()$ and $q_2()$. It updates
  - with probability 0.5

    $$q_1(s,a) \leftarrow q_1(s,a) + \alpha(r_t + \gamma q_2(s', argmax_{a' \in \mathcal{A}} \ q_1(s', a')) - q_1(s,a)), \ \ \forall s, a$$

  - with probability 0.5

    $$q_2(s,a) \leftarrow q_2(s,a) + \alpha(r_t + \gamma q_1(s', argmax_{a' \in \mathcal{A}} \ q_2(s', a')) - q_2(s,a)), \ \ \forall s, a$$

  - Policy is improved using $q_1(\cdot, \cdot) + q_2(\cdot, \cdot)$.
- By using multiple Q-functions, it is known to have less variance.

Write pol_eval_dbl_Q()

```
pol_eval_Q <- function(sample_step, q_s_a, alpha) {
  s <- sample_step[1]
  a <- sample_step[2]
  r <- sample_step[3] %>% as.numeric()
  s_next <- sample_step[4]
  q_s_a[s,a] <- q_s_a[s,a] + alpha*(r+max(q_s_a[s_next,])-q_s_a[s,a]) # change here
  return(q_s_a)
}
pol_eval_dbl_Q <- function(sample_step, q_s_a_1, q_s_a_2, alpha) {
  s <- sample_step[1]
  a <- sample_step[2]
  r <- sample_step[3] %>% as.numeric()
  s_next <- sample_step[4]
  if (runif(1) < 0.5) { # update q_s_a_1
    q_s_a_1[s,a] <- q_s_a_1[s,a] +
      alpha*(r+q_s_a_2[s_next, which.max(q_s_a_1[s_next,])]-q_s_a_1[s,a]) # change here
  } else { # update q_s_a_2
    q_s_a_2[s,a] <- q_s_a_2[s,a] +
      alpha*(r+q_s_a_1[s_next, which.max(q_s_a_2[s_next,])]-q_s_a_2[s,a]) # change here
  }
  return(list(q_s_a_1, q_s_a_2))
}
```

Double Q-learning

```r
num_ep <- 10^5
beg_time <- Sys.time() # change below
q_s_a_1 <- q_s_a_init; q_s_a_2 <- q_s_a_init
pi <- pi_50
exploration_rate <- 1
for (epi_i in 1:num_ep) {
  s_now <- "0"
  while (s_now != "70") {
    sample_step <- simul_step(pi, s_now, P_normal, P_speed, R_s_a)
    q_s_a <- pol_eval_dbl_Q(sample_step, q_s_a_1, q_s_a_2, alpha = max(1/epi_i, 0.05)) # change here
    q_s_a_1 <- q_s_a[[1]]; q_s_a_2 <- q_s_a[[2]] # change here
    if (epi_i %% 100 == 0) {
      pi <- pol_imp(pi, q_s_a_1+q_s_a_2, epsilon = exploration_rate) # change here
    }
    s_now <- sample_step[4]
    exploration_rate <- max(exploration_rate*0.9995, 0.001)
  }
```

```r
print(Sys.time()-beg_time)
```

```
## Time difference of 10.58 secs
```

```r
t(pi)
```

```
##   0 10 20 30 40 50 60 70
## n 0  1  0  1  1  0  1  1
## s 1  0  1  0  0  1  0  0
```

```r
t((q_s_a_1 + q_s_a_2)/2)
```

```
##        0     10     20     30     40     50     60 70
## n -5.492 -4.539 -3.843 -2.778 -1.703 -1.891 -1.000  0
## s -5.071 -4.839 -3.410 -3.284 -1.840 -1.627 -1.695  0
```

Exercise 1

Implement double-Q learning, and feel free to try different schemes for the number of iterations and exploration decaying scenarios.

```
"It's not that I'm so smart, it's just that I stay with problems longer. - A. Einstein"
```