Machine Learning 2ST129 26605 HT2023 Assignment 8

Anonymous Student

January 7, 2024

Contents

General Information

- Time used for reading 3 hours:
- Time used for basic assignment 14 hours:
- Time used for extra assignment NA:
- Good with lab: It was fun to implement different algorithms and see how they performed. Also i felt that i learned the basic idea of reinforcement learning.
- Things improve with lab: Not necessarily an improvement, but I think it could have been fun to implement some larger task rather than just many smaller tasks and have to task focused to be more of a program rather than just some individual functions.

```r
#Libraries
 library(tidyverse)
 library(xtable)
 library(tensorflow)
 library(keras)
```

1 Task 1: Bandits

1.1 Task 1.1: stationary bandits

Here we implement a stationary bandit function, which takes an action and returns a reward. The reward is sampled from a normal distribution with mean $\mu = q_a^*$ and standard deviation $\sigma = 1$ and where $q_a^* = (1 - 62, 1.2, 0.7, 0.72, 2.03)$.

```r
#do the same as staionary bandit but check that a is an integer between 1 and 5
stationary_bandit <- function(a, t=NULL){
  q_star <- c(1.62, 1.2, 0.7, 0.72, 2.03)
  if(a %in% 1:5){
    reward <- rnorm(1, mean = q_star[a], sd = 1)
    return(reward)
  } else{
    print("a must be an integer between 1 and 5")
  }
}
```

```r
#test the function
set.seed(4711)
stationary_bandit(2)
```

```
[1] 3.019735
```

```r
stationary_bandit(1)
```

```
[1] 2.99044
```

```r
R <- numeric(1000)
for(i in 1: 1000){
  R[i] <- stationary_bandit(3)
}
mean(R)
```

```
[1] 0.6819202
```

1.2 Task 1.2: non-stationary bandits

Next we implement a non-stationary bandit function. For all actions 2 to 5, it works as the stationary reward function, but for a = 1 it returns a reward based on normal(-3 +0.01t, 1).

```r
nonstationary_bandit <- function(a, t){
  q_star <- c(1.62, 1.2, 0.7, 0.72, 2.03)
  if(a %in% 1:5){
    if(a == 1){
      reward <- rnorm(1, mean = -3 + 0.01*t, sd = 1)
      return(reward)
    } else{
      reward <- rnorm(1, mean = q_star[a], sd = 1)
      return(reward)
    }
  } else{
    print("a must be an integer between 1 and 5")
  }
```

```
}
```

```
set.seed(4711)
nonstationary_bandit(2, t = 1)
```

```
[1] 3.019735
```

```
nonstationary_bandit(2, t = 1000)
```

```
[1] 2.57044
```

```
nonstationary_bandit(1, t = 1)
```

```
[1] -1.793682
```

```
nonstationary_bandit(1, t = 1000)
```

```
[1] 6.593121
```

1.3 Task 1.3: greedy algorithm

Here we implement a greedy algorithm that always chooses the greedy action according to the equation: $A_t \doteq \underset{a}{\text{argmax}} Q_t(a)$. The function takes a vector of initial estimates Q_1 and a reward function, and then run the bandit for 1000 steps and return four things: the rewards R_t for all steps, the mean reward for all 1000 steps, the value estimates $Q_t(a)$ and the total number of choices of each action, $Nt(a)$.

```
greedy_algorithm <- function(Q1, bandit, n_steps = 1000) {
  num_actions <- length(Q1)
  all_rewards <- numeric(n_steps)
  Q_t <- numeric(num_actions)
  N_t <- rep(0, num_actions)

  for (t in 1:n_steps) {
    action_t <- which.max(Q1)

    if (identical(bandit, stationary_bandit)) {
      reward_t <- stationary_bandit(action_t)
    } else if (identical(bandit, nonstationary_bandit)) {
      reward_t <- nonstationary_bandit(action_t, t)
    } else {
      stop("Invalid bandit function")
    }

    # Update estimates
    N_t[action_t] <- N_t[action_t] + 1
    Q1[action_t] <- Q1[action_t] + (reward_t - Q1[action_t]) / N_t[action_t]

    # Save reward
    all_rewards[t] <- reward_t
    Q_t[action_t] <- Q1[action_t]
  }

  mean_reward <- mean(all_rewards)

  return(list(Qt = Q_t, Nt = N_t, R_bar = mean_reward, Rt = all_rewards))
}
```

```
set.seed(4711)
Q1 <- rep(0, 5)
lapply(greedy_algorithm(Q1, stationary_bandit), head)
```

```
$Qt
```

```
[1] 1.602744 0.000000 0.000000 0.000000 0.000000

$Nt
[1] 1000    0    0    0    0

$R_bar
[1] 1.602744

$Rt
[1] 3.439735 2.990440 2.816318 1.213121 1.009021 0.111088
```

1.4 Task 1.4: ϵ-greedy algorithm

Now we implement an ϵ-greedy algorithm that chooses the greedy action with probability $1 - \epsilon$ and a random action with probability ϵ. It is based on the same greedy algorithm as before, but now the parameter `espilon` is included as an argument.

```
epsilon_algorithm <- function(Q1, bandit, epsilon, n_steps=1000) {
  #instantiating initial variables
  num_actions <- length(Q1)
  all_rewards <- numeric(n_steps)
  Q_t <- numeric(num_actions)
  N_t <-rep(0, num_actions)

  for (t in 1:n_steps) {
    if (runif(1) < epsilon) {
      action_t <- sample(1:num_actions, 1)
    } else {
      action_t <- which.max(Q1)
    }

     if (identical(bandit, stationary_bandit)) {
      reward_t <- stationary_bandit(action_t)
    } else if (identical(bandit, nonstationary_bandit)) {
      reward_t <- nonstationary_bandit(action_t, t)
    } else {
      stop("Invalid bandit function")
    }

    # Update estimates and save reward
    N_t[action_t] <- N_t[action_t] + 1
    Q1[action_t] <- Q1[action_t] + (reward_t - Q1[action_t]) / N_t[action_t]

    all_rewards[t] <- reward_t
    Q_t[action_t] <- Q1[action_t]
  }

  mean_reward <- mean(all_rewards)

  return(list(Qt = Q_t, Nt = N_t, R_bar = mean_reward, R_t = all_rewards))
}
set.seed(4711)
Q1 <- rep(0, 5)
lapply(epsilon_algorithm(Q1, stationary_bandit, epsilon = 0.1), head)
```

```
$Qt
[1] 1.6093183 1.1491153 0.7382012 0.5975228 2.0272784

$Nt
```

```
[1] 234  17  19  16 714

$R_bar
[1] 1.867178

$R_t
[1] 1.7725176 2.8163182 0.6335086 2.0175494 0.1021861 2.0943547
```

```
set.seed(4712)
Q1 <- rep(0, 5)
lapply(epsilon_algorithm(Q1, stationary_bandit, epsilon = 0.1), head)
```

```
$Qt
[1] 1.2660095 1.4034524 0.4066058 0.4252434 1.9752285

$Nt
[1]  21  16  14  20 929

$R_bar
[1] 1.898226

$R_t
[1] 0.2196340 1.8486270 0.6191028 0.5085077 3.8471272 1.2287379
```

1.5 Task 1.5: ϵ-nonstationary algorithm

Here we implement an ϵ-greedy algorithm for the non-stationary bandit. It is based on the same ϵ-greedy algorithm as before, but now we have an additional parameter, α as a constant step-size parameter.

```
nonstationary_algorithm <- function(Q1, bandit, epsilon, alpha, n_steps=1000) {
 #instantiating initial variables
 num_actions <- length(Q1)
 all_rewards <- numeric(n_steps)
 Q_t <- numeric(num_actions)
 N_t <-rep(0, num_actions)

 for (t in 1:n_steps) {
   if (runif(1) < epsilon) {
     action_t <- sample(1:num_actions, 1)
   } else {
     action_t <- which.max(Q1)
   }

     if (identical(bandit, stationary_bandit)) {
     reward_t <- stationary_bandit(action_t)
   } else if (identical(bandit, nonstationary_bandit)) {
     reward_t <- nonstationary_bandit(action_t, t)
   } else {
     stop("Invalid bandit function")
   }

   N_t[action_t] <- N_t[action_t] + 1
   #here we only changed how q is updated with alpha
   Q1[action_t] <- Q1[action_t] + alpha*(reward_t - Q1[action_t])

   all_rewards[t] <- reward_t
   Q_t[action_t] <- Q1[action_t]
 }
```

```
 mean_reward <- mean(all_rewards)

 return(list(Qt = Q_t, Nt = N_t, R_bar = mean_reward, R_t = all_rewards))
}
set.seed(4711)
Q1 <- rep(0, 5)
lapply(nonstationary_algorithm(Q1, stationary_bandit, epsilon = 0.1, alpha=0.2), head)
```

```
$Qt
[1] 1.414101 1.293336 1.237085 0.517214 2.134716

$Nt
[1] 302  16  19  16 647

$R_bar
[1] 1.840128

$R_t
[1] 1.7725176 2.8163182 0.6335086 2.0175494 0.5221861 2.0943547
```

```
set.seed(4712)
Q1 <- rep(0, 5)
lapply(nonstationary_algorithm(Q1, stationary_bandit, epsilon = 0.1, alpha=0.2), head)
```

```
$Qt
[1] 0.9785426 1.4271567 0.1610887 0.2170131 2.3148956

$Nt
[1]  56  17  14  19 894

$R_bar
[1] 1.884356

$R_t
[1] 0.2196340 1.8486270 0.6191028 0.5085077 3.8471272 1.2287379
```

1.6 Task 1.6: multiple simulations

Next we run each algorithm 500 times and compute the mean of the rewards for each of the algorithm. For the stationary bandit we use *alpha* $= 0.5$ and 0.9, with $\epsilon = 0.1$. For the ϵ-bandit, we use $\epsilon = 0.1$

```
set.seed(4711)
Q1 <- rep(0, 5)
stationary_greedy_res<- replicate(500,
                            greedy_algorithm(Q1, stationary_bandit, n_steps=1000),
                          simplify = FALSE)
nonstationary_greedy_res <- replicate(500,
                              greedy_algorithm(Q1, nonstationary_bandit, n_steps=1000),
                            simplify = FALSE)

stationary_epsilon_res <- replicate(500,
                              epsilon_algorithm(Q1, stationary_bandit,
                                                epsilon = 0.1, n_steps=1000),
                            simplify = FALSE)

non_stationary_epsilon_res <- replicate(500,
                                  epsilon_algorithm(Q1, nonstationary_bandit,
                                       epsilon = 0.1, n_steps=1000),
                                simplify = FALSE)
```

```r
stationary_alpha_res1 <- replicate(500,
                                   nonstationary_algorithm(Q1, stationary_bandit,
                                       epsilon = 0.1, alpha=0.5, n_steps=1000),
                                   simplify = FALSE)

nonstationary_alpha_res1 <- replicate(500,
                                      nonstationary_algorithm(Q1, nonstationary_bandit,
                                          epsilon = 0.1, alpha=0.5, n_steps=1000),
                                      simplify = FALSE)

stationary_alpha_res2 <- replicate(500, nonstationary_algorithm(Q1, stationary_bandit,
                                        epsilon = 0.1, alpha=0.5, n_steps=1000),
                                   simplify = FALSE)

nonstationary_alpha_res2 <- replicate(500, nonstationary_algorithm(Q1, nonstationary_bandit,
                                           epsilon = 0.1, alpha=0.9, n_steps=1000),
                                      simplify = FALSE)

#epsilon bandit


#return the mean result for all 500 iterations
bandit_means <- function(bandit_results) {
  return(mean(sapply(bandit_results, function(x) x$R_bar)))
}

bandit_means(stationary_greedy_res)

[1] 1.594351

bandit_means(nonstationary_greedy_res)

[1] 1.166828

bandit_means(stationary_epsilon_res)

[1] 1.892233

bandit_means(non_stationary_epsilon_res)

[1] 2.011043

bandit_means(stationary_alpha_res1)

[1] 1.771884

bandit_means(nonstationary_alpha_res1)

[1] 2.957633

bandit_means(stationary_alpha_res2)

[1] 1.770182

bandit_means(nonstationary_alpha_res2)

[1] 2.867637
```

1.7 1.7 conclusions

Based on the results from task 1.7, The greedy algorithm on the non-stationary bandit performs worst, with a mean reward of 1.16. The non-stationary algorithm with $\alpha = 0.9$ performs best, with a mean reward of 2.96. The reason for this is that the greedy algorithm does not explore enough, and therefore does not find the best action. The non-stationary algorithm with $\alpha = 0.9$ performs best because the ϵ-part of the algorithm makes it explores enough to improve in order to find the optimal action, and

therefore finds the best action, while the α-part makes it have a "memory" such that it gives more weight to the most recent rewards, and therefore it can adapt to the non-stationary bandit.

1.8 1.8 average reward per step

Here we plot the average reward per step for the 500 runs for the worst and best algorithms as concluded from the previous section based on the output from R_t

```
#plot the average reward per step for the 500 runs for the worst and best
#algorithms as concluded from the previous section based on the output from $R_t$

#greedy algorithm on non-stationary bandit
greedy_nonstationary <- nonstationary_greedy_res[[1]]$Rt
for(i in 2:500){
  greedy_nonstationary <- cbind(greedy_nonstationary,
                                nonstationary_greedy_res[[i]]$Rt)
}
greedy_nonstationary <- rowMeans(greedy_nonstationary)

#non-stationary algorithm with alpha = 0.5
nonstationary_alpha <- nonstationary_alpha_res1[[1]]$R_t
for(i in 2:500){
  nonstationary_alpha <- cbind(nonstationary_alpha,
                               nonstationary_alpha_res1[[i]]$R_t)
}
nonstationary_alpha <- rowMeans(nonstationary_alpha)

plot(greedy_nonstationary, type = "l", col = "red",
     xlab = "steps", ylab = "average reward per step",
     main = "Average reward per step for the 500 runs for the worst and best algorithms",
     xlim = c(0, 1000), ylim = c(-3, 8))

lines(nonstationary_alpha, col = "blue")

legend("topleft", legend = c("greedy algorithm on non-stationary bandit",
                "non-stationary algorithm with alpha = 0.5 and epsilon = 0.1"),
       col = c("red", "blue"), lty = 1:2, cex = 0.8)
```

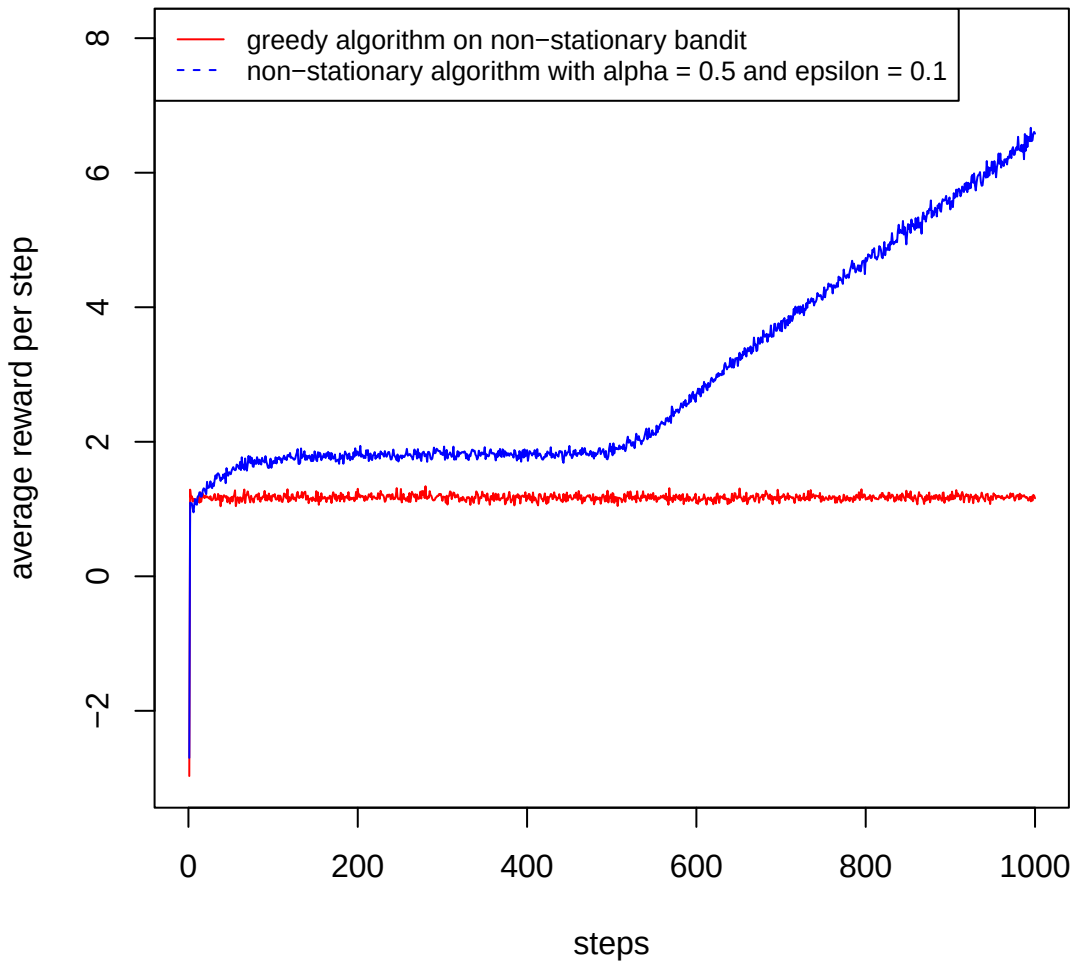

Figure 1: Average performance of best and worst algorithms

2 Task 2 Markov Decision Processes

Here we implement a markov decision process where the agent will do a decision A_t based on the current stat S_t as well as a reward R_t, and then return a new state S_{t+1} and a new reward R_{t+1}, using the recycling example.

2.1 Task 2.1: Recycle Robot

First we use the function for the MDP from the uuml package.

```
library(uuml)
mdp <- recycling_mdp(0.5, 0.8, 0.1, 1)
```

2.2 Task 2.2: always_search_policy

Now to implement a function `always_search_policy` which uses the MPD from the previous task and run the MDP for 1000 steps. The policy is that the agent always chooses to search and it returns the the return divided by the number of time steps and times in each state, and the robot start in the state

high. the function `always_search_poliicy` only takes the mdp as an argument, whereas the function `recycling_mdp` takes the parameters $\alpha, \beta, \gamma, r_{wait}, r_{search}$ and n_{steps} as arguments.

```
always_search_policy <-function(mdp, n_steps=1000){
  #initializing variables
  #the action will always be search for each iteration
states <- c("high", "low")
  N_t <- c(0,0)
  names(N_t) <- states
  rewards <- numeric(n_steps)
  cur_state <- "high"
  action <- "search"

  for (i in 1:n_steps) {
    row_index <- which(mdp$s == cur_state & mdp$a == action)
    p_next <- mdp$p_s_next[row_index]

    state_next <- sample(x = states, size = 1, prob = p_next)

    N_t[cur_state] <- N_t[cur_state] + 1
    reward_index <- which(mdp$s == cur_state & mdp$a == action & mdp$s_next == state_next)

    rewards[i] <- mdp$r_s_a_s_next[reward_index]

    cur_state <- state_next
  }

  return(list("Nt" = N_t, "R_bar" = mean(rewards)))

}
set.seed(4711)
always_search_policy(mdp)
```

```
$Nt
high  low
 288  712

$R_bar
[1] 0.452
```

2.3 Task 2.3: charge_when_low_policy

Now to implement a function `charge_when_low_policy` which uses the MPD from the previous task and run the MDP for 1000 steps. The policy is that the agent always chooses to charge when the state is low and always search if the energy level is high. The function returns the same output structure as the previous function. Hence it is basically the same function, but with an added if statement in the for loop so that the action is not always in search mode.

```
#function based on the charge_when_low_policy.
charge_when_low_policy <-function(mdp, n_steps=1000){
  #initializing variables
  N_t <- c(0,0)
  states <- c("high", "low")
  names(N_t) <- states
  rewards <- numeric(n_steps)
  cur_state <- "high"


  for (i in 1:n_steps) {
    if(cur_state == "high"){
```

```
      action <- "search"
    } else{
      action <- "recharge"
    }
    row_index <- which(mdp$s == cur_state & mdp$a == action)
    p_next <- mdp$p_s_next[row_index]

    state_next <- sample(x = states, size = 1, prob = p_next)

    N_t[cur_state] <- N_t[cur_state] + 1
    reward_index <- which(mdp$s == cur_state & mdp$a == action & mdp$s_next == state_next)

    rewards[i] <- mdp$r_s_a_s_next[reward_index]

    cur_state <- state_next
  }

  return(list("Nt" = N_t, "R_bar" = mean(rewards)))

}
set.seed(4711)
charge_when_low_policy(mdp)
```

```
$Nt
high  low
 669  331

$R_bar
[1] 0.669
```

2.4 Task 2.4 compare policies

Lastly we compare the two policies for MDP with $\alpha = \beta = 0.9, and \alpha = \beta = 0.4$ with $r_{\text{wait}} = 0.1, r_{\text{search}} = 1$ and $n_{\text{steps}} = 1000$.

```
#function that combines both policies given the parameters and returns the R_bar for both policies
compare_policies <- function(alpha, beta, r_wait, r_search, n_steps = 1000){
  mdp <- recycling_mdp(alpha, beta, r_wait, r_search)
  always_search <- always_search_policy(mdp, n_steps)$R_bar
  charge_when_low <- charge_when_low_policy(mdp, n_steps)$R_bar
  results <- c(always_search, charge_when_low)
  names(results) <- c("always_search", "charge_when_low")
  return(results)
}

replicate_compare_policies <- function(alpha, beta, r_wait, r_search,
                                       n_steps = 1000, n_replicates = 20){
  multi_results <-replicate(n_replicates,
                            mapply(compare_policies, alpha=alpha, beta=beta,
                                   r_wait=r_wait, r_search = r_search,
                                   n_steps = n_steps, SIMPLIFY = TRUE),
                            simplify = FALSE)
  #each replication of the mapply returns a table where the rows are the policies
  #and the columns are the MDPs. Hence multi_results is a list containing
  #n_replicates number of tables as each element.

means <- Reduce('+', multi_results) / n_replicates
colnames(means) <- c("MDP_1", "MDP_2")
return(means)
```

```
}

set.seed(4711)
replicate_compare_policies(alpha = c(0.9, 0.4), beta = c(0.9, 0.4), r_wait = 0.1,
       r_search = 1, n_steps = 1000, n_replicates = 15) %>%
  round(digits = 2)
```

```
                MDP_1 MDP_2
always_search    0.79 -0.20
charge_when_low  0.91  0.63
```

In the output, MDP_1 corresponds to $\alpha = \beta = 0.9$ and MDP_2 corresponds to $\alpha = \beta = 0.4$. In general we see that MDP_1 performs better than MDP_2. We see that the policy charge_when_low_policy performs better than always_search_policy for both $\alpha = \beta = 0.9$ and $\alpha = \beta = 0.4$. The reason for this is that the policy charge_when_low_policy is more efficient, as it only charges when the energy level is low, and therefore it can search more efficiently. The best one was the charge_when_low_policy for $\alpha = \beta = 0.9$ with mean R_bar $= 0.9$.