Advanced Machine Learning

Lab 3: Reinforcement Learning

Jakob Berggren

2023-10-02

Q-Learning

First off, the lab asks to complete the implementation of the Q-learning algorithm in the template file RL_Lab1.R. The code is available in its entirety at the end of this report.

To complete the implementation, three functions needed to be implemented. First, the GreedyPolicy function was implemented by simply returning the action with the maximum Q-value in the Q-table.

```
GreedyPolicy <- function(x, y) {

  # Get a greedy action for state (x,y) from q_table.
  #
  # Args:
  #   x, y: state coordinates.
  #   q_table (global variable): a HxWx4 array containing Q-values for each state-action pair.
  #
  # Returns:
  #   An action, i.e. integer in {1,2,3,4}.

  return (which.max(q_table[x, y, ]))
}
```

Next, the EpsilonGreedyPolicy was to be implemented. This includes so called "exploration", by doing random actions with a certain probability determined by the parameter epsilon. Otherwise, the regular Greedy Policy is used.

```
EpsilonGreedyPolicy <- function(x, y, epsilon) {

  # Get an epsilon-greedy action for state (x,y) from q_table.
  #
  # Args:
  #   x, y: state coordinates.
  #   epsilon: probability of acting randomly.
  #
  # Returns:
  #   An action, i.e. integer in {1,2,3,4}.

  if (runif(1) < epsilon) {
    # Do random explorations. This samples a random Action
    return (sample(1:4, 1))
  } else {
    return(GreedyPolicy(x ,y))
  }
}
```

Last, the Q-learning function was implemented. This is done by implementing a loop which then follows a pattern as such:

1. Follow the policy. This gets the next action by following the Epsilon Greedy Policy.
2. Execute the action to get the next state. The next state is computed by the Transition Model function which was given in the template file.
3. Get the reward for the new state from the reward map.

Then, the Q-table is updated using the Q-learning formula. Here the temporal correction term is also computed which is summed to get the episode correction. This is then iterated to get the final Q-table.

```
q_learning <- function(start_state, epsilon = 0.5, alpha = 0.1, gamma = 0.95,
                       beta = 0) {
```

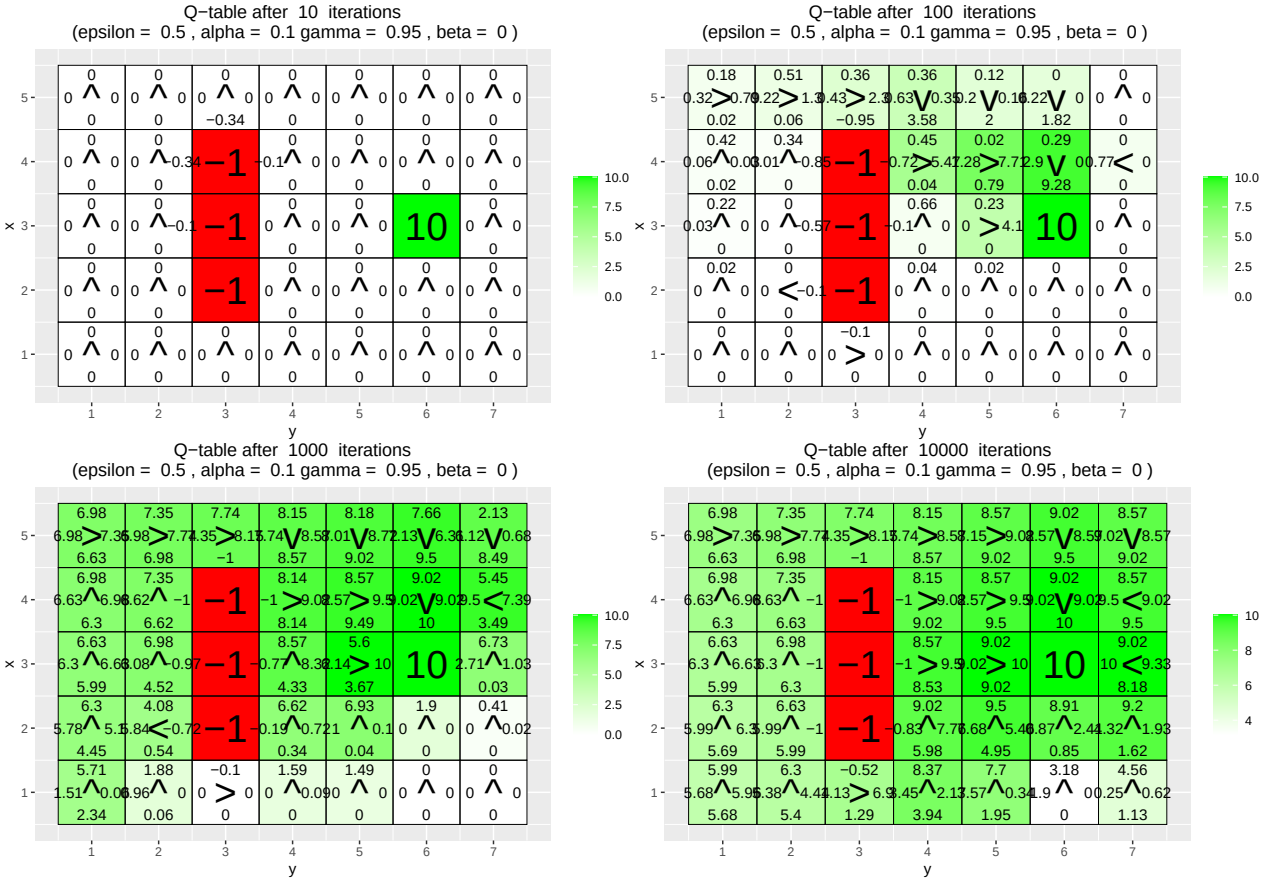
```
  # Perform one episode of Q-learning. The agent should move around in the
  # environment using the given transition model and update the Q-table.
  # The episode ends when the agent reaches a terminal state.
  #
  # Args:
  #   start_state: array with two entries, describing the starting position of the agent.
  #   epsilon (optional): probability of acting greedily.
  #   alpha (optional): learning rate.
  #   gamma (optional): discount factor.
  #   beta (optional): slipping factor.
  #   reward_map (global variable): a HxW array containing the reward given at each state.
  #   q_table (global variable): a HxWx4 array containing Q-values for each state-action pair.
  #
  # Returns:
  #   reward: reward received in the episode.
  #   correction: sum of the temporal difference correction terms over the episode.
  #   q_table (global variable): Recall that R passes arguments by value. So, q_table being
  #   a global variable can be modified with the superassigment operator <<-.

  x <- start_state[1]
  y <- start_state[2]
  episode_correction <- 0 # Init episode correction

  repeat {
    # Follow policy, execute action, get reward.

    # Follow policy.
    action <- EpsilonGreedyPolicy(x, y, epsilon)

    # Execute action. Compute new state using transition model.
    new_state <- transition_model(x, y, action, beta)

    # Get reward for new state from reward map.
    reward <- reward_map[new_state[1], new_state[2]]

    # Compute correction and update Q-table with Q-learning formula.
    correction <- reward + gamma * max(q_table[new_state[1], new_state[2], ]) - q_table[x, y, action]
    q_table[x, y, action] <<- q_table[x, y, action] + alpha * correction # Update Q-table.

    # Update episode correction
    episode_correction <- episode_correction + correction

    if(reward != 0) {
      # End episode.
      return (c(reward,episode_correction))
    }

    # Update state
    x <- new_state[1]
    y <- new_state[2]
  }
}
```

Next, the code implemented was used to conduct experiments on three different environments.

Environment A

The first environment will use H = 5, W = 7, a reward of 10 in state (3,6) and a reward of -1 in states (2,3), (3,3) and (4,3). The rewards are specified using a reward map in the form of a matrix with one entry for each state. States with no reward will simply have a matrix entry of 0.

Running 10000 episodes of Q-learning with $\epsilon = 0.5$, $\beta = 0$, $\alpha = 0.1$ and $\gamma = 0.95$ yielded the following output.



1. What has the agent learned after the first 10 episodes?

After the first episodes, the agent has not learned much. We know the Agent starts in (3,1), and by 10 iterations it has learned that being close to the squares with negative reward will likely have a negative outcome.

2. Is the final greedy policy (after 10000 episodes) optimal for all states, i.e. not only for the initial state ? Why / Why not?

No, it does not seem to be optimal for each state. For example, one can see that in state (1,2), the greedy policy wants to go up, but it would be more beneficial to go to the right since this would more quickly reach the 10-reward "goal".

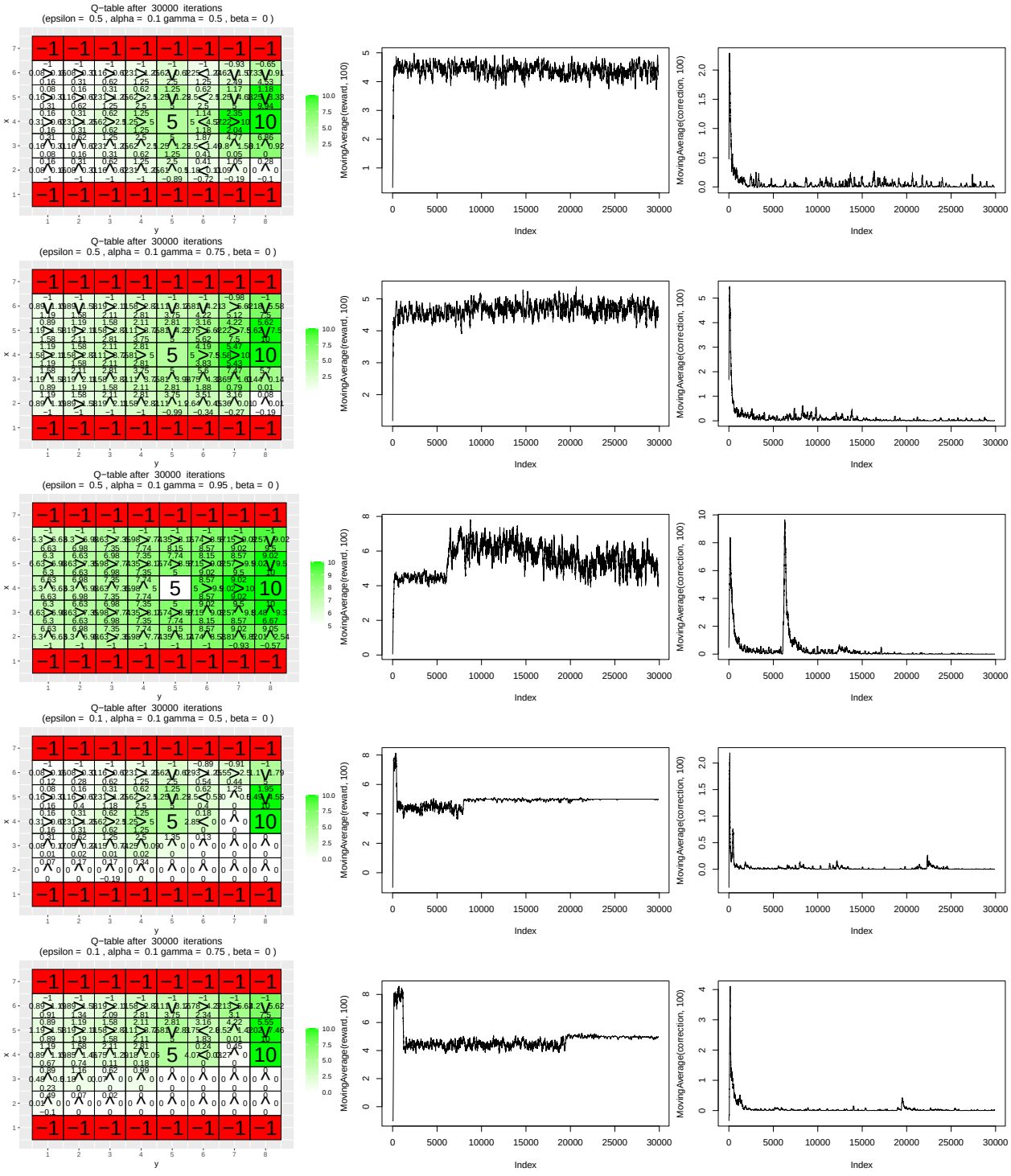
3. Do the learned values in the Q-table reflect the fact that there are multiple paths (above and below the negative rewards) to get to the positive reward ? If not, what could be done to make it happen?

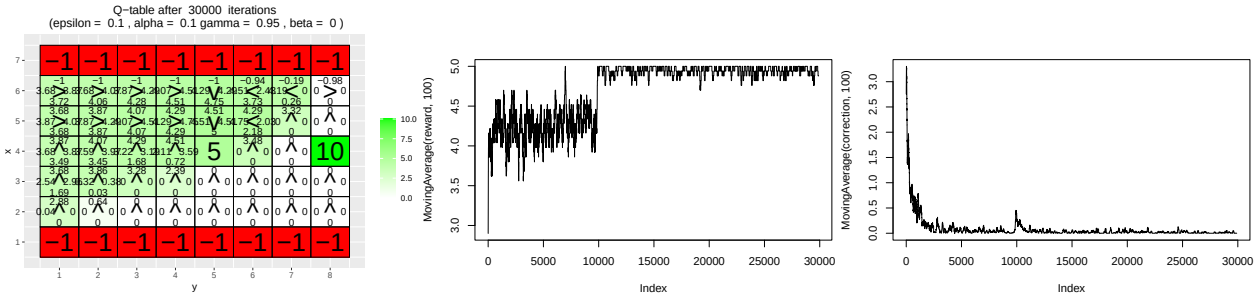
Kind off, however, the paths below the negative rewards have lower q-values and thus the Agent would favor the above path. One thing that could be done is increase epsilon in the early iterations which would encourage more exploration.

Environment B

Environment B is a 7×8 environment where the top and bottom rows have negative rewards. In this environment, the agent starts each episode in the state (4, 1). There are two positive rewards, of 5 and 10. The reward of 5 is easily reachable, but the agent has to navigate around the first reward in order to find the reward worth 10.

The objective is to investigate how the ϵ and γ parameters affect the learned policy by running 30000 episodes of Q-learning with $\epsilon = 0.1, 0.5$, $\gamma = 0.5, 0.75, 0.95$, $\beta = 0$ and $\alpha = 0.1$. The results of this can be seen below.





Epsilon

Epsilon promotes exploration by increasing the probability of the agent making a random action. This can benefit the agent since it will potentially find high reward paths more quickly. However, since the agent does more random actions, convergence of the algorithm can be slower. In this case, where the 10 point reward is further away and thus not as easy to reach, the random explorations is helping the agent finding the higher reward instead of getting stuck around the lesser reward.

Gamma

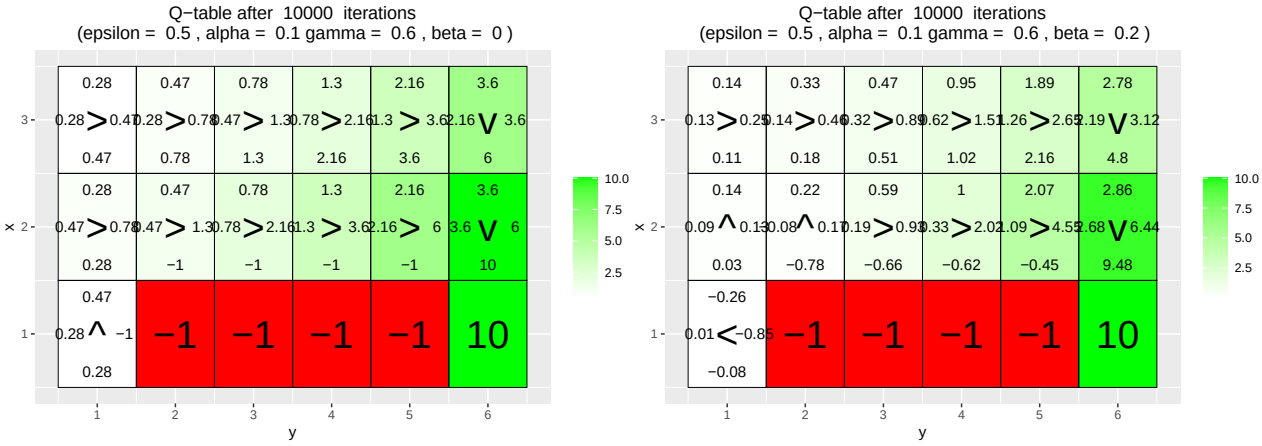
The gamma value is a type of discount factor, which can be used alter by which degree the agent favors future rewards vs immediate rewards. A high gamma value would make the Agent favor future rewards more. This is very apparent in the plots, where the higher gamma value agent clearly favors the 10 point reward over the 5 point reward.

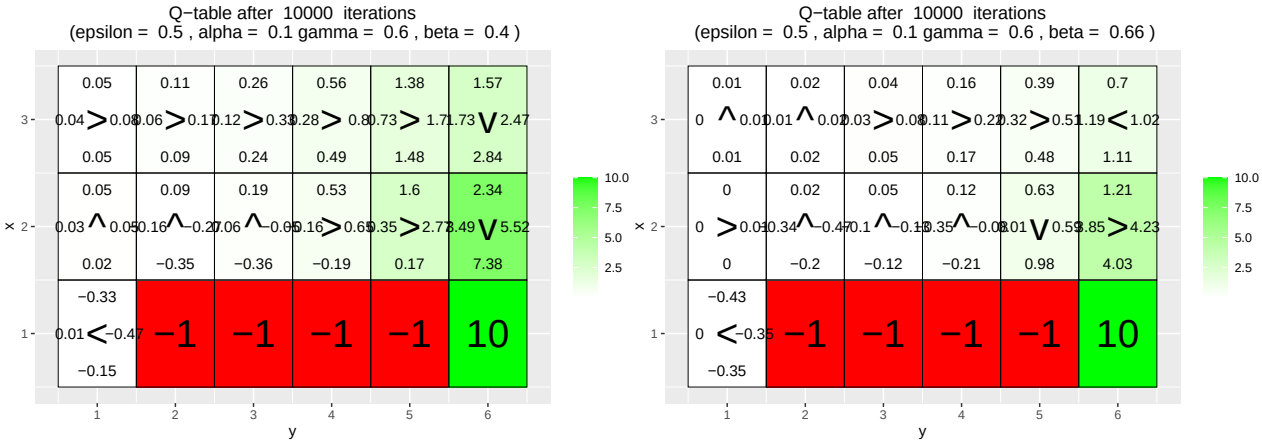
Overall, it is important to find a balance between the Agents willingness to explore and favoring of immediate rewards, which can be done by altering the epsilon and gamma values.

Environment C

Environment C is a smaller 3×6 environment. Here the agent starts each episode in the state (1,1).

Investigate how the β parameter affects the learned policy by running 10000 episodes of Q-learning with $\beta = 0,\ 0.2,\ 0.4,\ 0.66$, $\epsilon = 0.5$, $\gamma = 0.6$ and $\alpha = 0.1$. Results of the experiment can be seen below.





Beta

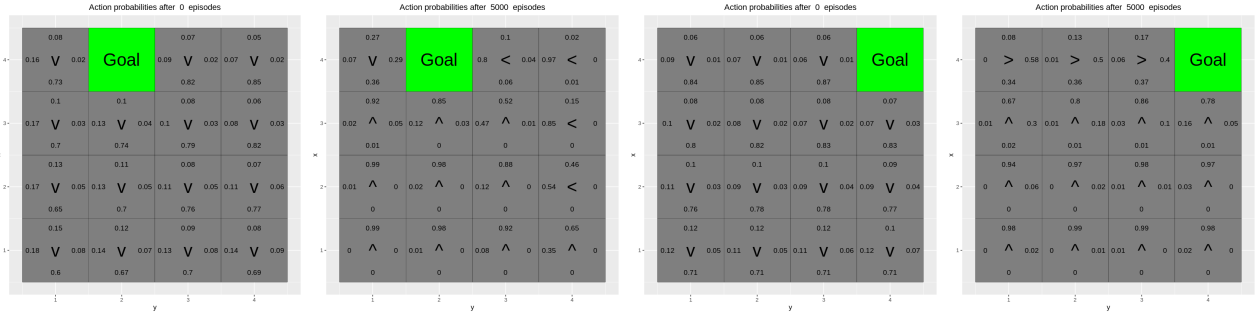
The beta value works as a "slipping factor". This means that the Agent will execute an action other than the intended with a certain probability, determined by beta. This can very easily be seen in the plots above, where the low beta value agent is fine with going close to the negative rewards, while the high beta value agent learns to stay further away from the negative rewards (since it might "slip") even if this means it will take a longer path to the final reward.

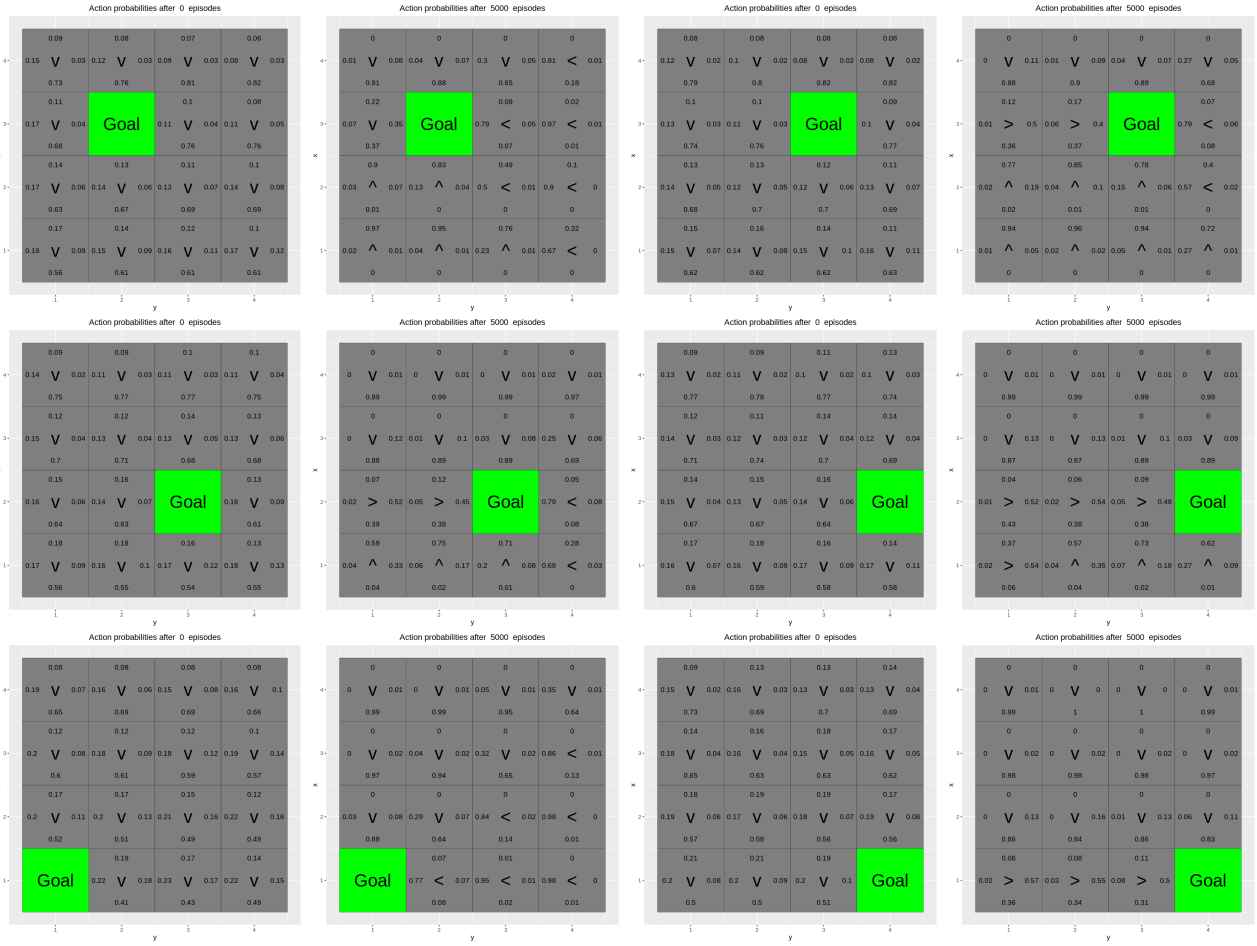
REINFORCE

The next part of the lab explores the REINFORCE algorithm. The file RL_Lab2_Colab.ipynb contains the implementation and result of the running code. Two different environments are to be studied. These will work in a 4 x 4 grid, where the agent will learn to navigate to a random goal position in the grid. When the agent reaches the goal, it will receive a reward of 5.

Environment D

Environment D will use eight goal positions for training and, then, validate the learned policy on the remaining eight possible goal positions. Each training episode uses a random goal position from train_goals. The initial position for the episode is also chosen at random. Results can be seen in the figures below.





1. Has the agent learned a good policy? Why / Why not?

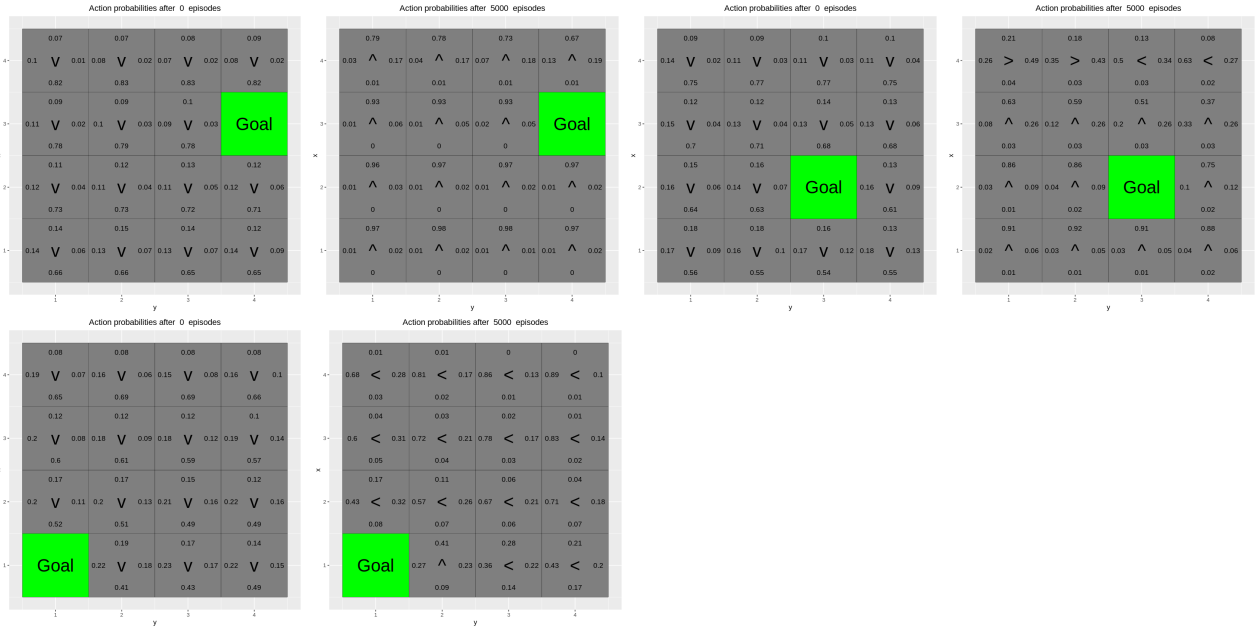
Yes, the agent has learned a good policy. Looking at the output figures, one can see that the agent has found the optimal policy for most of the states. In some cases, the policy states similar probability for two directions even one of them is clearly getting the agent closer to the goal. So, in that sense, the policy is good, but might not be optimal.

2. Could you have used the Q-learning algorithm to solve this task?

No, the Q-learning algorithm would not be optimal to solve this task, since the goal position is random.

Environment E

In environment E the goals for training are all from the top row of the grid. The validation goals are three positions from the rows below. The results can be seen below.



1. Has the agent learned a good policy? Why / Why not?

No, the agent has not learned a good policy. Looking at the output figures, the agent is clearly not finding the goal. The figures also indicate that the agent prefers to move upwards, which is a result of the training data, where the goals are all in the top of the grid. The training data did not yield a good general policy, but one that would only be beneficial if the goal were at the top of the grid.

2. If the results obtained for environments D and E differ, explain why.

The results obtained for environments D and E do differ. As mentioned, the training data for D and E is very different, which results in certain bias. Environment E has very clear bias towards its training data, and it is clear that the agent does not solve the task very well. Environment D has much less bias and the agent is able to adapt to the validation data and find the goal.

Code - Q-learning

```
# By Jose M. Peña and Joel Oskarsson.
# For teaching purposes.
# jose.m.pena@liu.se.

##########################################################################
# Q-learning
##########################################################################

# install.packages("ggplot2")
# install.packages("vctrs")
library(ggplot2)

arrows <- c("^", ">", "v", "<")
action_deltas <- list(c(1,0), # up
                      c(0,1), # right
                      c(-1,0), # down
                      c(0,-1)) # left

vis_environment <-
  function(iterations = 0,
           epsilon = 0.5,
           alpha = 0.1,
           gamma = 0.95,
           beta = 0) {

  # Visualize an environment with rewards.
  # Q-values for all actions are displayed on the edges of each tile.
  # The (greedy) policy for each state is also displayed.
  #
  # Args:
  #   iterations, epsilon, alpha, gamma, beta (optional): for the figure title.
  #   reward_map (global variable): a HxW array containing the reward given at each state.
  #   q_table (global variable): a HxWx4 array containing Q-values for each state-action pair.
  #   H, W (global variables): environment dimensions.

  df <- expand.grid(x=1:H,y=1:W)
  foo <- mapply(function(x,y) ifelse(reward_map[x,y] == 0,q_table[x,y,1],NA),df$x,df$y)
  df$val1 <- as.vector(round(foo, 2))
  foo <- mapply(function(x,y) ifelse(reward_map[x,y] == 0,q_table[x,y,2],NA),df$x,df$y)
  df$val2 <- as.vector(round(foo, 2))
  foo <- mapply(function(x,y) ifelse(reward_map[x,y] == 0,q_table[x,y,3],NA),df$x,df$y)
  df$val3 <- as.vector(round(foo, 2))
  foo <- mapply(function(x,y) ifelse(reward_map[x,y] == 0,q_table[x,y,4],NA),df$x,df$y)
  df$val4 <- as.vector(round(foo, 2))
  foo <- mapply(function(x,y)
    ifelse(reward_map[x,y] == 0,arrows[GreedyPolicy(x,y)],reward_map[x,y]),df$x,df$y)
  df$val5 <- as.vector(foo)
  foo <- mapply(function(x,y) ifelse(reward_map[x,y] == 0,max(q_table[x,y,]),
                                     ifelse(reward_map[x,y]<0,NA,reward_map[x,y])),df$x,df$y)
  df$val6 <- as.vector(foo)

  print(ggplot(df,aes(x = y,y = x)) +
```

```
          scale_fill_gradient(low = "white", high = "green", na.value = "red", name = "") +
          geom_tile(aes(fill=val6)) +
          geom_text(aes(label = val1),size = 4,nudge_y = .35,na.rm = TRUE) +
          geom_text(aes(label = val2),size = 4,nudge_x = .35,na.rm = TRUE) +
          geom_text(aes(label = val3),size = 4,nudge_y = -.35,na.rm = TRUE) +
          geom_text(aes(label = val4),size = 4,nudge_x = -.35,na.rm = TRUE) +
          geom_text(aes(label = val5),size = 10) +
          geom_tile(fill = 'transparent', colour = 'black') +
          ggtitle(paste("Q-table after ",iterations," iterations\n",
                        "(epsilon = ",epsilon,", alpha = ",alpha,"gamma = ",
                        gamma,", beta = ",beta,")")) +
          theme(plot.title = element_text(hjust = 0.5)) +
          scale_x_continuous(breaks = c(1:W),labels = c(1:W)) +
          scale_y_continuous(breaks = c(1:H),labels = c(1:H)))

}

GreedyPolicy <- function(x, y) {

  # Get a greedy action for state (x,y) from q_table.
  #
  # Args:
  #   x, y: state coordinates.
  #   q_table (global variable): a HxWx4 array containing Q-values for each state-action pair.
  #
  # Returns:
  #   An action, i.e. integer in {1,2,3,4}.

  # Your code here.

  # sample uniformly from the set of actions with maximal Q-value.
  return (which.max(q_table[x, y, ]))

}

EpsilonGreedyPolicy <- function(x, y, epsilon) {

  # Get an epsilon-greedy action for state (x,y) from q_table.
  #
  # Args:
  #   x, y: state coordinates.
  #   epsilon: probability of acting randomly.
  #
  # Returns:
  #   An action, i.e. integer in {1,2,3,4}.

  # Your code here.
  if (runif(1) < epsilon) {
    # Do random explorations. This samples a random Action
    return (sample(1:4, 1))
  } else {
    return(GreedyPolicy(x ,y))
  }
```

```
}

transition_model <- function(x, y, action, beta) {

  # Computes the new state after given action is taken. The agent will follow the action
  # with probability (1-beta) and slip to the right or left with probability beta/2 each.
  #
  # Args:
  #   x, y: state coordinates.
  #   action: which action the agent takes (in {1,2,3,4}).
  #   beta: probability of the agent slipping to the side when trying to move.
  #   H, W (global variables): environment dimensions.
  #
  # Returns:
  #   The new state after the action has been taken.

  delta <- sample(-1:1, size = 1, prob = c(0.5*beta,1-beta,0.5*beta))
  final_action <- ((action + delta + 3) %% 4) + 1
  foo <- c(x,y) + unlist(action_deltas[final_action])
  foo <- pmax(c(1,1),pmin(foo,c(H,W)))

  return (foo)
}

q_learning <- function(start_state, epsilon = 0.5, alpha = 0.1, gamma = 0.95,
                       beta = 0) {

  # Perform one episode of Q-learning. The agent should move around in the
  # environment using the given transition model and update the Q-table.
  # The episode ends when the agent reaches a terminal state.
  #
  # Args:
  #   start_state: array with two entries, describing the starting position of the agent.
  #   epsilon (optional): probability of acting greedily.
  #   alpha (optional): learning rate.
  #   gamma (optional): discount factor.
  #   beta (optional): slipping factor.
  #   reward_map (global variable): a HxW array containing the reward given at each state.
  #   q_table (global variable): a HxWx4 array containing Q-values for each state-action pair.
  #
  # Returns:
  #   reward: reward received in the episode.
  #   correction: sum of the temporal difference correction terms over the episode.
  #   q_table (global variable): Recall that R passes arguments by value. So, q_table being
  #   a global variable can be modified with the superassigment operator <<-.

  # Your code here.
  x <- start_state[1]
  y <- start_state[2]
  episode_correction <- 0 # Init episode correction

  repeat {
    # Follow policy, execute action, get reward.
```

```
    # Follow policy.
    action <- EpsilonGreedyPolicy(x, y, epsilon)

    # Execute action. Compute new state using transition model.
    new_state <- transition_model(x, y, action, beta)

    # Get reward for new state from reward map.
    reward <- reward_map[new_state[1], new_state[2]]

    # Compute error and update Q-table with Q-learning formula.
    correction <- reward + gamma * max(q_table[new_state[1], new_state[2], ]) - q_table[x, y, action]
    q_table[x, y, action] <<- q_table[x, y, action] + alpha * correction # Update Q-table.

    # Update episode correction
    episode_correction <- episode_correction + correction


    if(reward != 0) {
      # End episode.
      return (c(reward,episode_correction))
    }

    # Update state
    x <- new_state[1]
    y <- new_state[2]
  }
}

#########################################################################
# Q-Learning Environments
#########################################################################

# Environment A (learning)

H <- 5
W <- 7

reward_map <- matrix(0, nrow = H, ncol = W)
reward_map[3,6] <- 10
reward_map[2:4,3] <- -1

q_table <- array(0,dim = c(H,W,4))

vis_environment()

for(i in 1:10000) {
  foo <- q_learning(start_state = c(3,1))

  if(any(i==c(10,100,1000,10000)))
    vis_environment(i)
}

# Environment B (the effect of epsilon and gamma)
```

```r
H <- 7
W <- 8

reward_map <- matrix(0, nrow = H, ncol = W)
reward_map[1,] <- -1
reward_map[7,] <- -1
reward_map[4,5] <- 5
reward_map[4,8] <- 10

q_table <- array(0,dim = c(H,W,4))

vis_environment()

MovingAverage <- function(x, n) {

  cx <- c(0,cumsum(x))
  rsum <- (cx[(n+1):length(cx)] - cx[1:(length(cx) - n)]) / n

  return (rsum)
}

for(j in c(0.5,0.75,0.95)) {
  q_table <- array(0,dim = c(H,W,4))
  reward <- NULL
  correction <- NULL

  for(i in 1:30000){
    foo <- q_learning(gamma = j, start_state = c(4,1))
    reward <- c(reward,foo[1])
    correction <- c(correction,foo[2])
  }

  vis_environment(i, gamma = j)
  plot(MovingAverage(reward,100),type = "l")
  plot(MovingAverage(correction,100),type = "l")
}

for(j in c(0.5,0.75,0.95)) {
  q_table <- array(0,dim = c(H,W,4))
  reward <- NULL
  correction <- NULL

  for(i in 1:30000){
    foo <- q_learning(epsilon = 0.1, gamma = j, start_state = c(4,1))
    reward <- c(reward,foo[1])
    correction <- c(correction,foo[2])
  }

  vis_environment(i, epsilon = 0.1, gamma = j)
  plot(MovingAverage(reward,100),type = "l")
  plot(MovingAverage(correction,100),type = "l")
}
```

```r
# Environment C (the effect of beta).

H <- 3
W <- 6

reward_map <- matrix(0, nrow = H, ncol = W)
reward_map[1,2:5] <- -1
reward_map[1,6] <- 10

q_table <- array(0,dim = c(H,W,4))

vis_environment()

for(j in c(0,0.2,0.4,0.66)) {
  q_table <- array(0,dim = c(H,W,4))

  for(i in 1:10000)
    foo <- q_learning(gamma = 0.6, beta = j, start_state = c(1,1))

  vis_environment(i, gamma = 0.6, beta = j)
}
```