Computer Lab 3
732A96 - Advanced Machine Learning

Luo Ying - `yinlu879`
José Méndez - `josme478`
Marcos F. Mourão - `marfr825`
Agustín Valencia - `aguva779`

October 9, 2020

Acknowledgements

All group members worked on their individual reports on their own and actively participated in the discussion of theory, coding, and in the writing of this collaborative report. The group agreed that the following represents the best answers from all individual reports.

In order to save knitting time the REINFORCE results were produced once and saved to be used while knitting later on.

1. Q-Learning

The file RL `Lab1.R` in the course website contains a template of the Q-learning algorithm. You are asked to complete the implementation. We will work with a gridworld environment consisting of $H \times W$ tiles laid out in a 2-dimensional grid. An agent acts by moving up, down, left or right in the grid-world. This corresponds to the following Markov decision process:

- State space : $\mathcal{S} = \{(x, y) \mid x \in \{1, \dots, H\}, y \in \{1, \dots, W\}\}$
- Action space : $\mathcal{A} = \{\text{up}, \text{down}, \text{left}, \text{right}\}$

Additionally, we assume state space to be fully observable. The reward function is a deterministic function of the state and does not depend on the actions taken by the agent. We assume the agent gets the reward as soon as it moves to a state. The transition model is defined by the agent moving in the direction chosen with probability $(1 - \beta)$. The agent might also slip and end up moving in the direction to the left or right of its chosen action, each with probability $\beta/2$. The transition model is unknown to the agent, forcing us to resort to model-free solutions. The environment is episodic and all states with a non-zero reward are terminal. Throughout this lab we use integer representations of the different actions: Up=1, right=2, down=3 and left=4.

Solution

We make use of the given code [1] which is also shown in Appendix A. We are asked to complete the `q_learning` function from the same provided code. For doing so it has been applied the following algorithm

Do repeat:

1. From current state s get an action a from ϵ-greedy policy
2. Compute next state s' using the given transition model
3. From new state get its reward R
4. Compute $Q(s', a')$
5. Compute correction $C = R + \gamma Q(s', a') - Q(s, a)$
6. Update Q-Table as $Q(s, a) \leftarrow Q(s, a) + \alpha \cdot C$
7. Episode correction : Episode Correction$_{new}$ = Episode Correction$_{old} + C$
8. If $R = 0$: break and return R and Episode Correction

The code implementation can be found in Appendix B.

2. Environment A

For our first environment, we will use $H = 5$ and $W = 7$. This environment includes a reward of 10 in state $(3, 6)$ and a reward of -1 in states $(2, 3)$, $(3, 3)$ and $(4, 3)$. We specify the rewards 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. The agent starts each episode in the state $(3, 1)$. The function `vis_environment` in the file `RL_Lab1.R` is used to visualize the environment and learned action values and policy. You will not have to modify this function, but read the comments in it to familiarize with how it can be used.

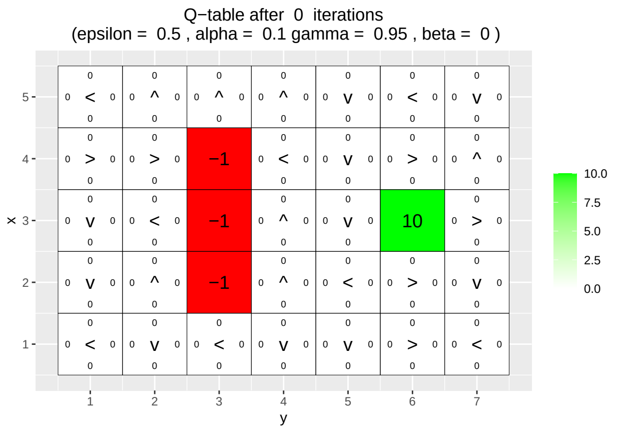
When implementing Q-learning, the estimated values of $Q(S, A)$ are commonly stored in a data-structured called Q-table. This is nothing but a tensor with one entry for each state-action pair. Since we have a $H \times W$ environment with four actions, we can use a 3D-tensor of dimensions $H \times W \times 4$ to represent our Q-table. Initialize all Q-values to 0. Run the function `vis environment` before proceeding further. Note that each non-terminal tile has four values. These represent the action values associated to the tile (state). Note also that each non-terminal tile has an arrow. This indicates the greedy policy for the tile (ties are broken at random).

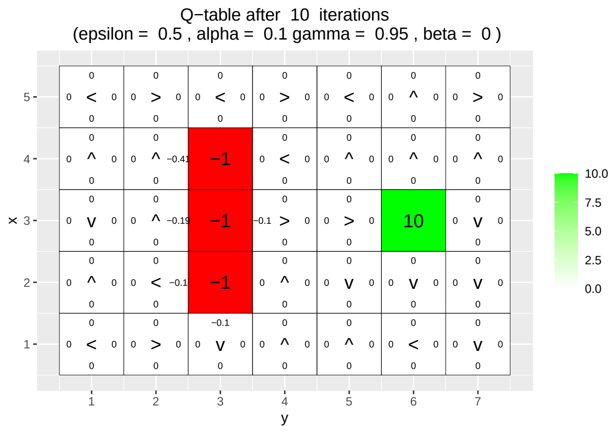
Your are requested to carry out the following tasks.

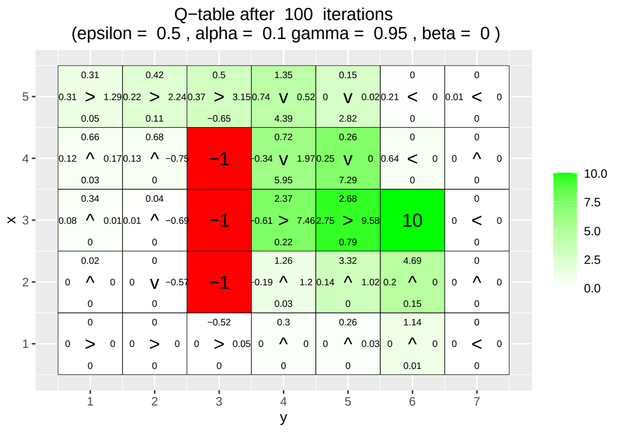
- Implement the greedy and ϵ-greedy policies in the functions `GreedyPolicy` and `EpsilonGreedyPolicy` of the file `RL_Lab1.R`. The functions should break ties at random, i.e. they should sample uniformly from the set of actions with maximal Q-value.
- Implement the Q-learning algorithm in the function `q_learning`of the file `RL_Lab1.R`. The function should run one episode of the agent acting in the environment and update the Q-table accordingly. The function should return the episode reward and the sum of the temporal-difference correction terms $R + \gamma \max_a Q(S', a) - Q(S, A)$ for all steps in the episode. Note that a transition model taking β as input is already implemented for you in the function `transition_model`.
- Run 10000 episodes of Q-learning with $\epsilon = 0.5$, $\alpha = 0.1$, $\gamma = 0.95$. To do so, simply run the code provided in the file `RL_Lab1.R`. The code visualizes the Q-table and a greedy policy derived from it after episodes 10, 100, 1000 and 10000. Answer the following questions:
  – What has the agent learned after the first 10 episodes ?
  – Is the final greedy policy (after 10000 episodes) optimal? Why / Why not ?
  – Does the agent learn that there are multiple paths to get to the positive reward ? If not, what could be done to make the agent learn this ?

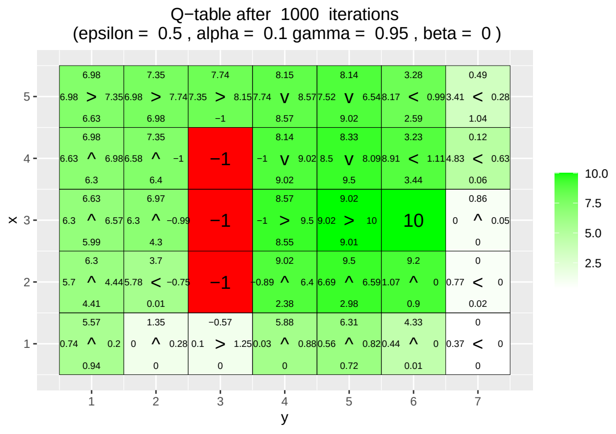
Solution

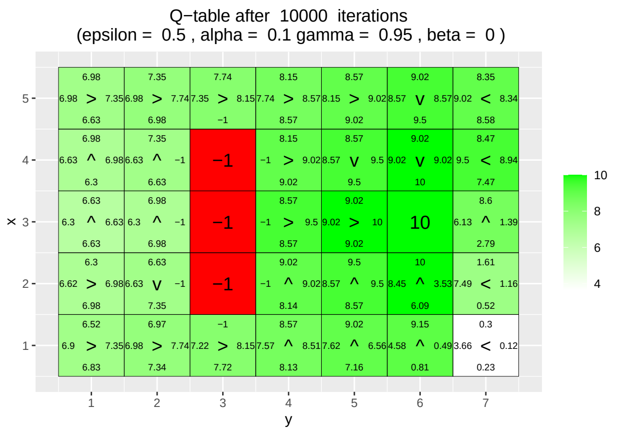
The policies are very similar in concept. They differ in terms of greediness, a pure greedy policy will always try to decide for the optimal action, which can lead into a lack of exploration of the environment when there are no more ties found while trying to get the goal. Whereas an ϵ-greedy policy allows an ϵ chance to the agent to be *adventurous* and explore randomly the environment instead of going for the *best* action in the Q-Table. Their code implementation can be found in Appendix C.











- What has the agent learned after the first 10 episodes?
After 10 episodes, the agent in the square world has barely learned where are the punishment blocks when approaching from the left. But it has still a lot to learn.

- Is the final greedy policy (after 10000 episodes) optimal? Why / Why not?
After 10,000 episodes we could say that the policy is optimal, because the agent "knows" exactly how to move around the punishments and where is the big reward.

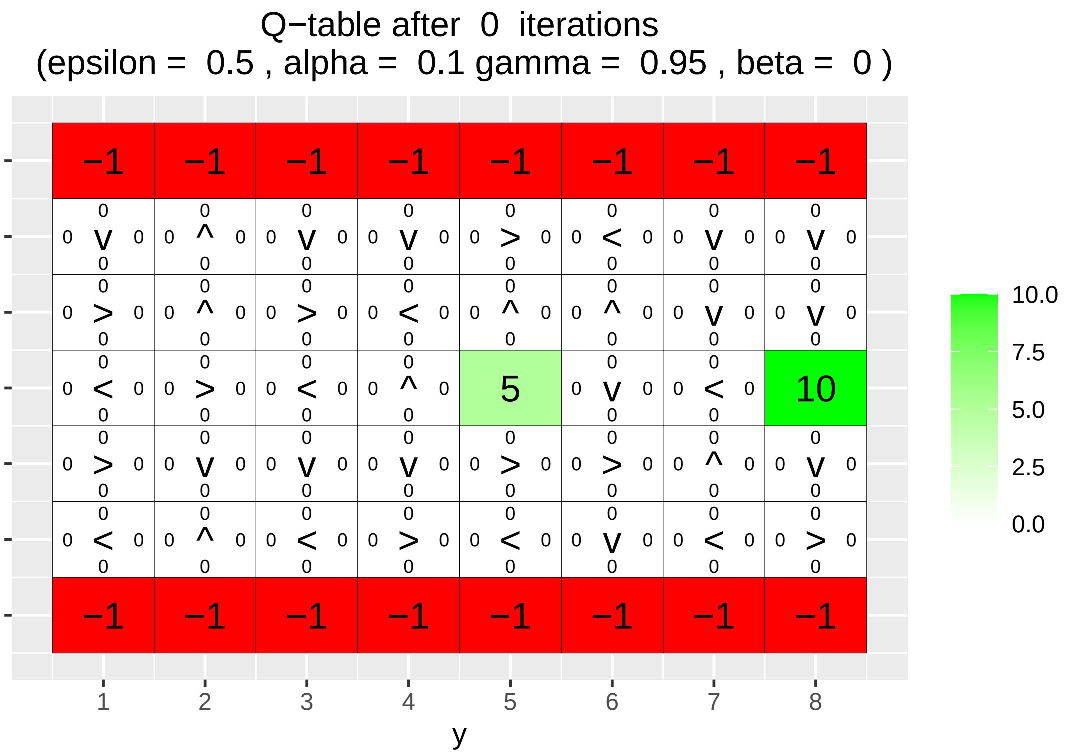
- Does the agent learn that there are multiple paths to get to the positive reward? If not, what could be done to make the agent learn this ?
The agent knows that there are multiple paths, but based on the greedy policy it will take more probably the path with bigger scores (which might be seen as local reward). Also in this environment, after 100 iterations, the agent knows where are the punishments, so we can add code to make the probabilities of taking the -1 punishment = 0.

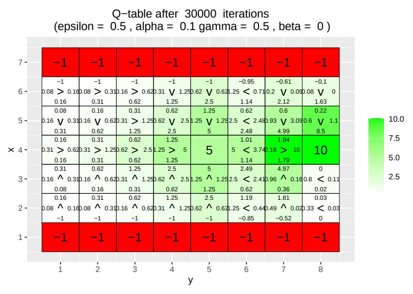
3. Environment B

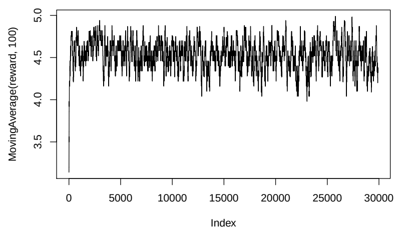
This is a 7×8 environment where the top and bottom rows have negative rewards. In this environment, the agent starts each episode in the state $(4, 1)$. There are two positive rewards, of 5 and 10. The reward of 5 is easily reachable, but the agent has to navigate around the first reward in order to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

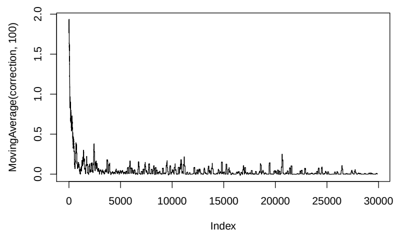
Your task is to investigate how the ϵ and γ parameters affect the learned policy by running 30000 episodes of Q-learning with $\epsilon = 0.1, 0.5$, $\gamma = 0.5, 0.75, 0.95$, $\beta = 0$ and $\alpha = 0.1$. To do so, simply run the code provided in the file `RL_Lab1.R` and explain your observations.

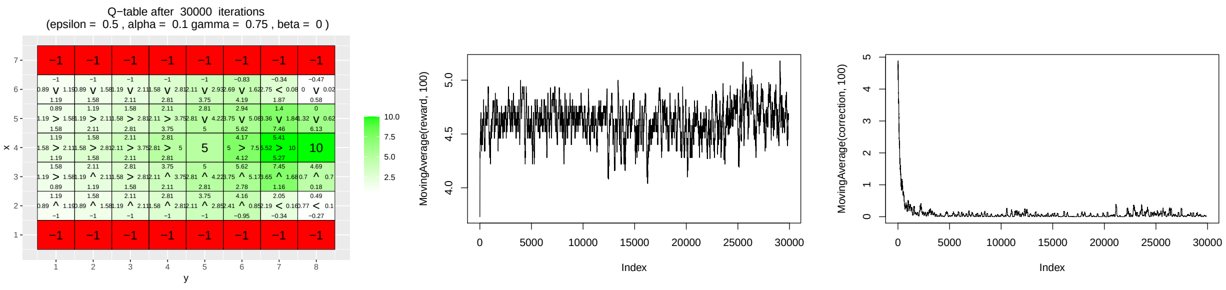
Solution

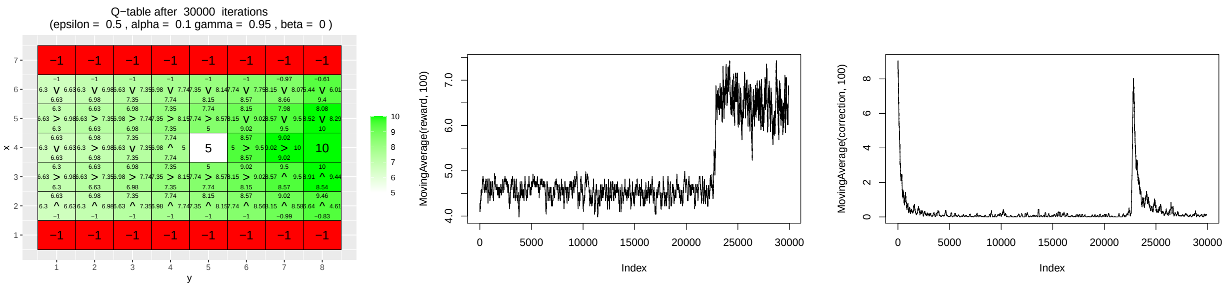


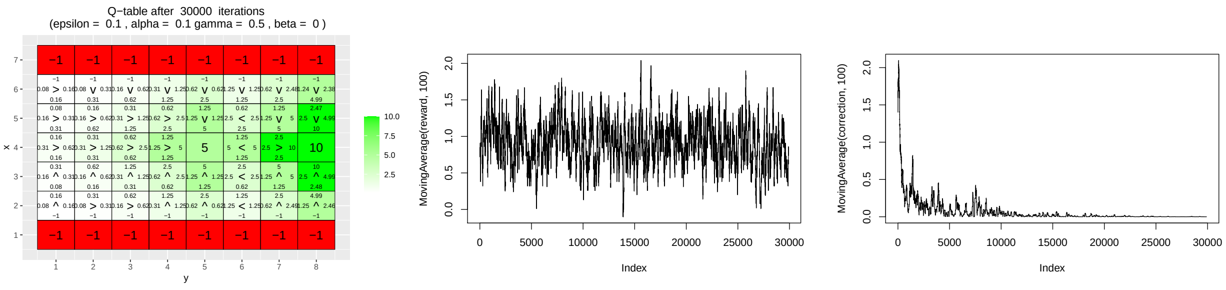


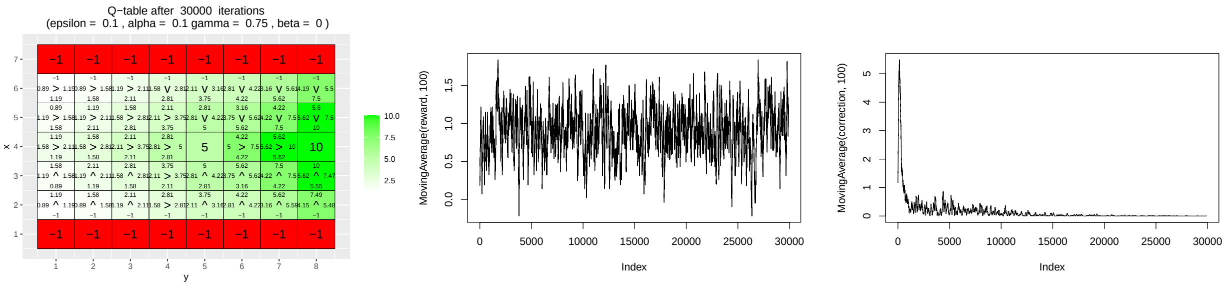


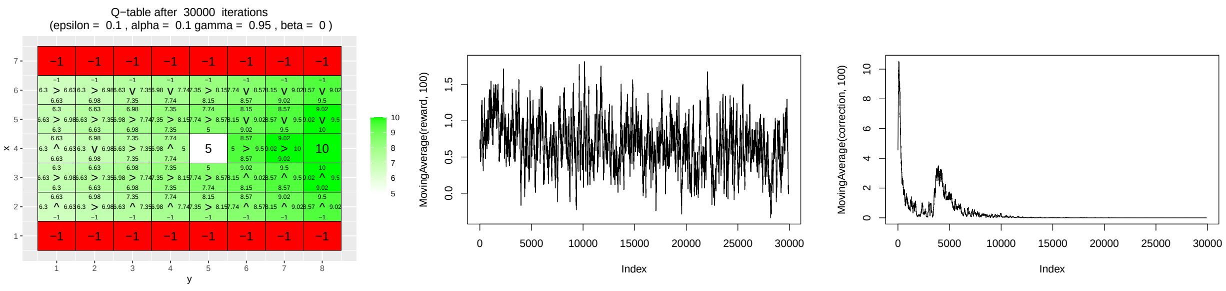












Tables	$\gamma = 0.5$	$\gamma = 0.75$	$\gamma = 0.95$
$\epsilon = 0.1$	After converge, reward almost stay at 5.	After converge, reward almost stay at 5.	After converge, reward almost stay at 5.
$\epsilon = 0.5$	Reward converged around 4.3-4.8.	Reward converged around 4.5-4.8.	After 23000 iterations, the reward converged around 6.2-7.

The ϵ parameters controls how exploratory the agent is at the cost of not always following the highest action value. This means that the robot will move more freely and actually could learn how to avoid the "bait" reward of 5. We can see this in all plots where the parameter is high $\epsilon = 0.5$.

The γ parameter control how immediate the agent seeks for rewards: lower values of gamma means heavily discounted future rewards (more penalization) dictating a "short term behavior", while higher γ values dictate a more "long term behavior" of the agent (less penalization).

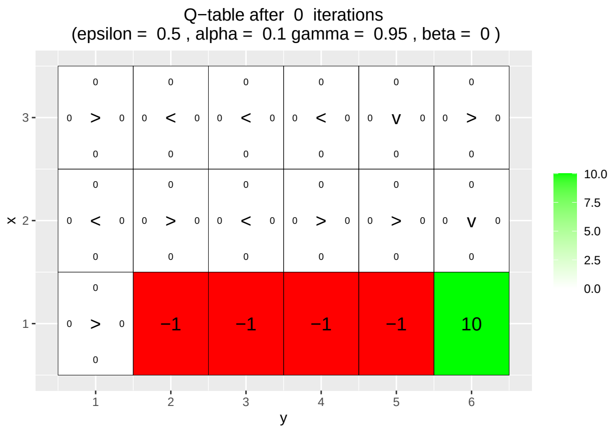
From the plots above we see that the agent trained with high values of $\epsilon = 0.5$ and $\gamma = 0.95$ is more exploratory and less keen on settling for earlier rewards, so we see this particular agent reach the "true" reward of 10. On the other hand, the agent with low value of epsilon $\epsilon = 0.1$ is more strict on following the maximum action value directions, even though higher values of γ makes it search for more delayed rewards, it is not enough to steer the agent away from the "bait".

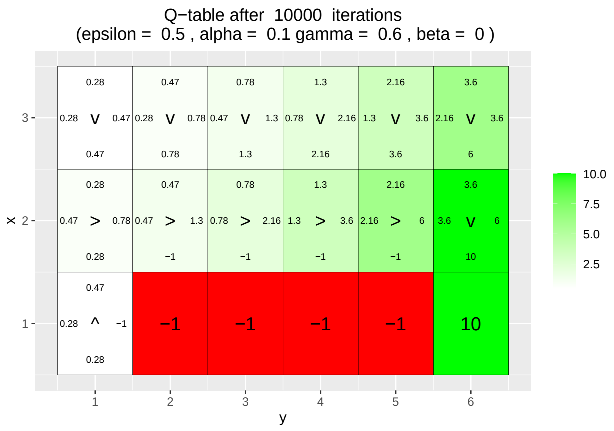
It is important to note that these parameters alone do not dictate how "good" or "bad" the agent is. This evaluation depends on the objective of the agent. Sometimes, the sacrifice of a bigger, but later reward is preferred in favour of a lower, earlier one.

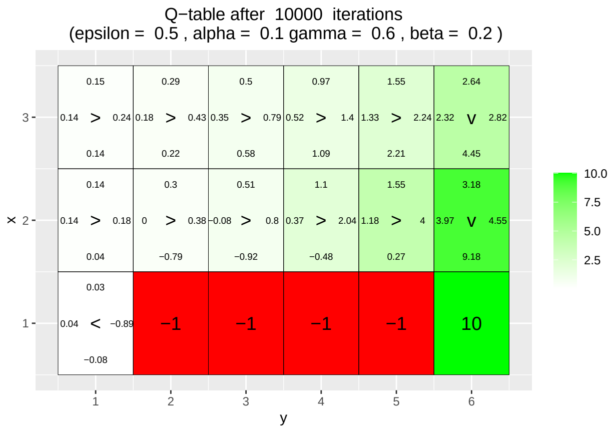
4. Environment C

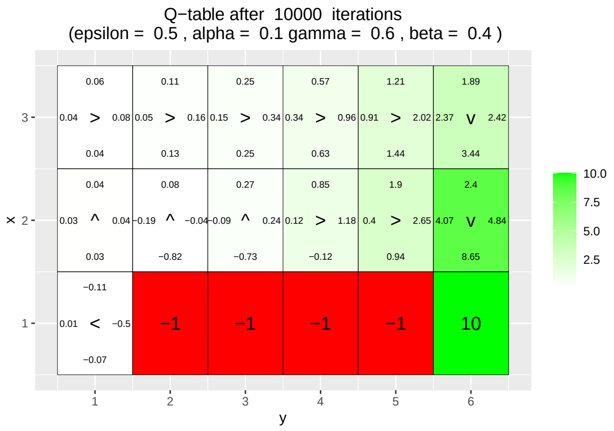
This is a smaller 3×6 environment. Here the agent starts each episode in the state $(1, 1)$. Your task is to investigate how the β parameter affects the learned policy by running 10000 episodes of Q-learning with $\beta = 0, 0.2, 0.4$, $\epsilon = 0.5$, $\gamma = 0.6$ and $\alpha = 0.1$. To do so, simply run the code provided in the file `RL_Lab1.R` and explain your observations.

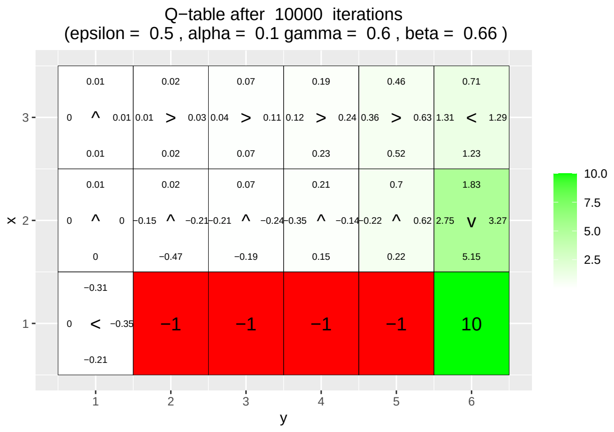
Solution





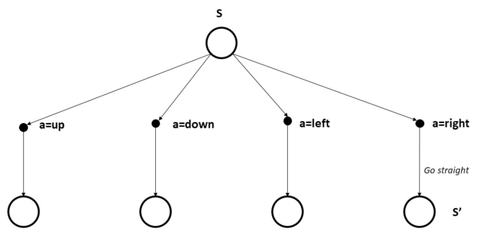
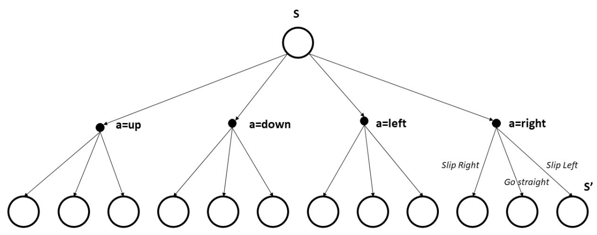






β controls the probability of slipping, this is, to be turned 90 degrees because of chance. In this scenario most of the cells in x = 1 are punishment cells. With $\beta = 0$, the agent does not slip in the path, and can walk normally over x = 2 and x = 3. But with a $\beta = 0.66$, the agent slips often and learns that on x = 2, it could slip to one of the punishment cells (and stop the iteration). So what it does is try to go safely over x3 and keep far away of x = 1

$\beta = 0$	$\beta \neq 0$

 s
a=up
a=down
a=left
a=right
Go straight
s'
	 s
	a=up
	a=down
	a=left
	a=right
	Slip Right
	Go straight
	Slip Left
	s'

5. REINFORCE

The file `RL_Lab2.R` in the course website contains an implementation of the REINFORCE algorithm. Your task is to run the code provided and answer some questions. Although you do not have to modify the code, you are advised to check it out to familiarize with it. The code uses the R package `keras` to manipulate neural networks.

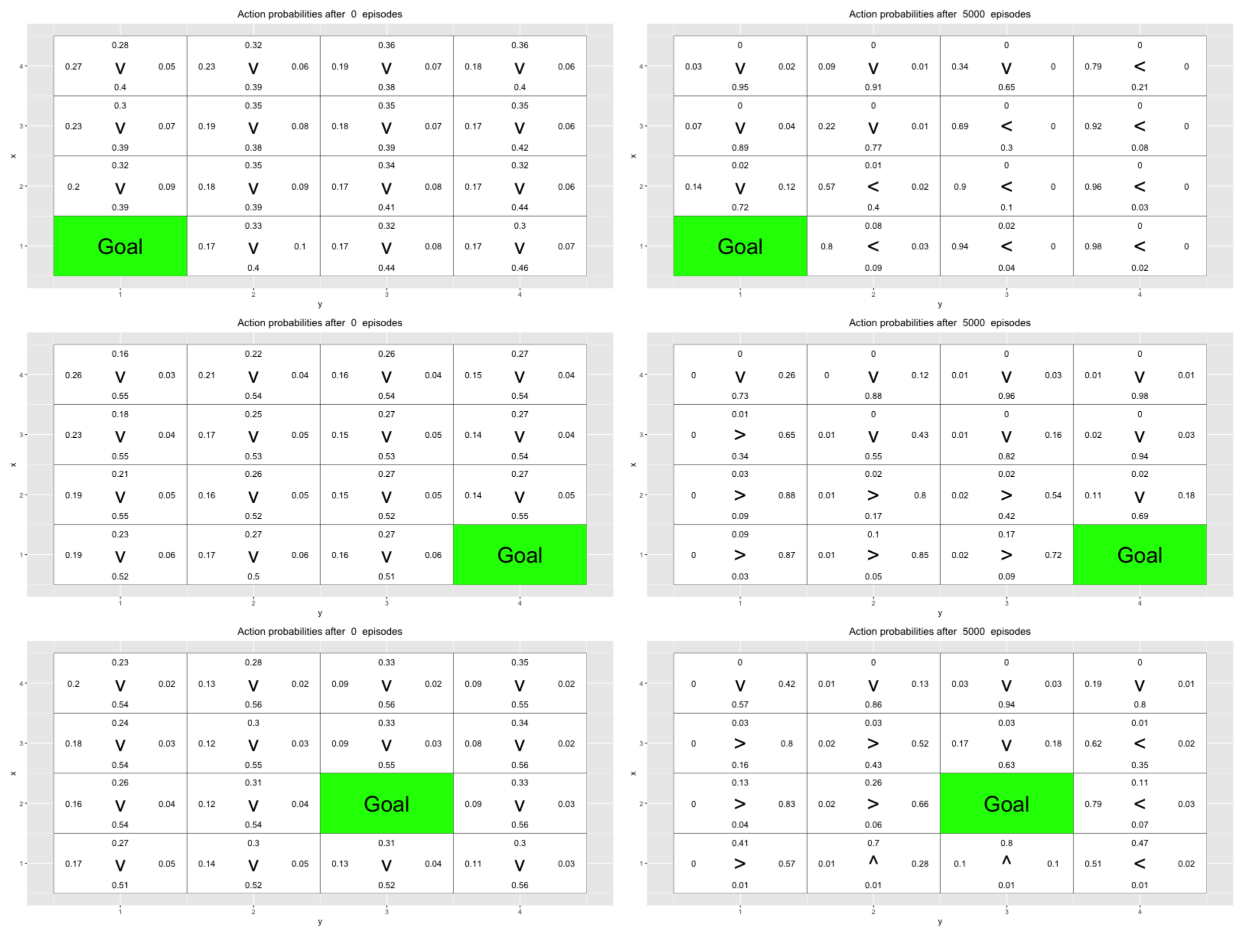
We will work with a 4×4 grid. We want the agent to learn to navigate to a random goal position in the grid. The agent will start in a random position and it will be told the goal position. The agent receives a reward of 5 when it reaches the goal. Since the goal position can be any position, we need a way to tell the agent where the goal is. Since our agent does not have any memory mechanism, we provide the goal coordinates as part of the state at every time step, i.e. a state consists now of four coordinates: Two for the position of the agent, and two for the goal position. The actions of the agent can however only impact its own position, i.e. the actions do not modify the goal position. Note that the agent initially does not know that the last two coordinates of a state indicate the position with maximal reward, i.e. the goal position. It has to learn it. It also has to learn a policy to reach the goal position from the initial position. Moreover, the policy has to depend on the goal position, because it is chosen at random in each episode. Since we only have a single non-zero reward, we do not specify a reward map. Instead, the goal coordinates are passed to the functions that need to access the reward function.

5.1 Environment D

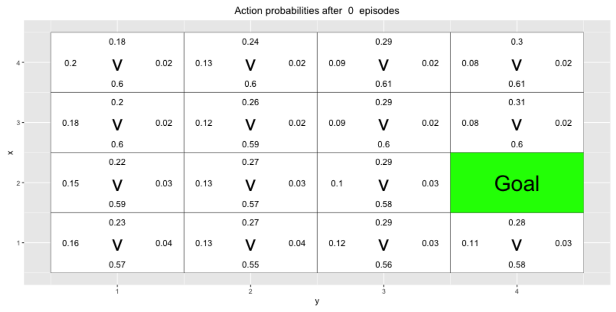
In this task, we will use eight goal positions for training and, then, validate the learned policy on the remaining eight possible goal positions. The training and validation goal positions are stored in the lists `train_goals` and `val_goals` in the code in the file `RL_Lab2.R`. You are requested to run the code provided, which runs the REINFORCE algorithm for 5000 episodes with $\beta = 0$ and $\gamma = 0.95$[1]. Each training episode uses a random goal position from train goals. The initial position for the episode is also chosen at random. When training is completed, the code validates the learned policy for the goal positions in val goals. This is done by with the help of the function vis prob, which shows the grid, goal position and learned policy. Note that each non-terminal tile has four values. These represent the action probabilities associated to the tile (state). Note also that each non-terminal tile has an arrow. This indicates the action with the largest probability for the tile (ties are broken at random). Finally, answer the following questions:

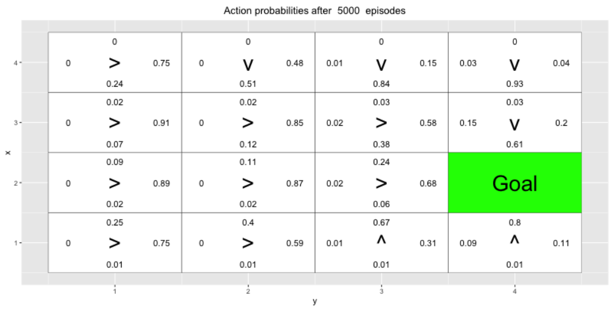
- Has the agent learned a good policy? Why / Why not ?
- Could you have used the Q-learning algorithm to solve this task ?

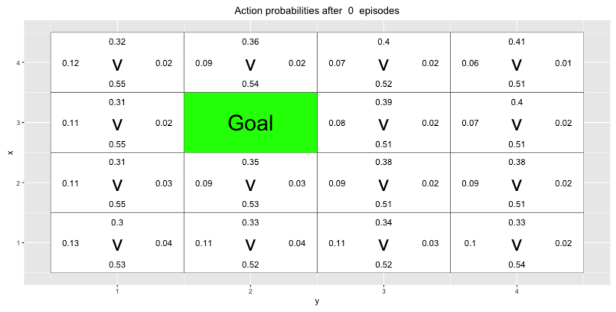
Solution

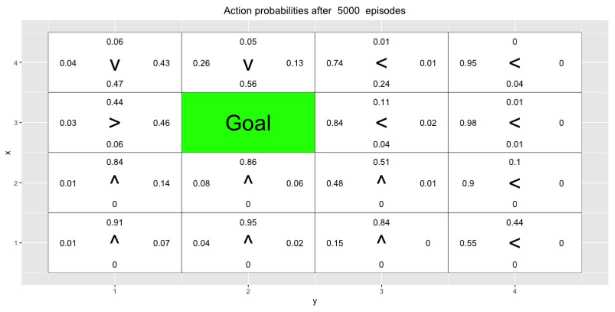


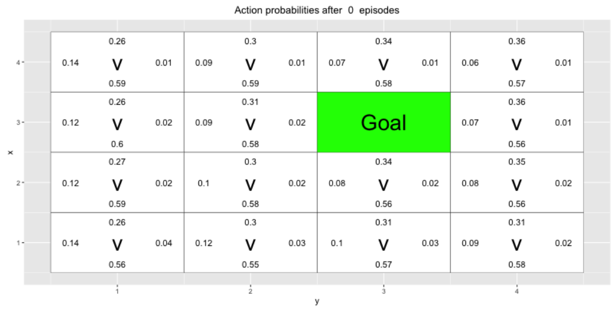
[1]Note that we are performing back-propagation in a neural network in each episode. Even though the network is small, this is quite computationally heavy and may take some time. Specifically, it takes around 12 wall-clock minutes in my laptop (Intel(R) Core(TM) i7-6600U CPU at 2.60GHz with 16.0 GB RAM and Windows 10). If someone finds a way to speed up the code, I would love to hear about it. The same code implemented in Python is faster. So, **keras** seems to get along better with Python than with R.

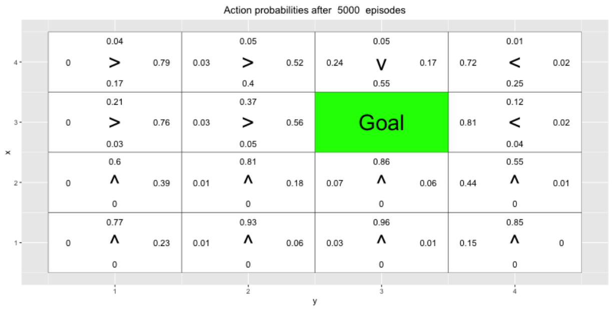

Action probabilities after 0 episodes
Goal
x
y

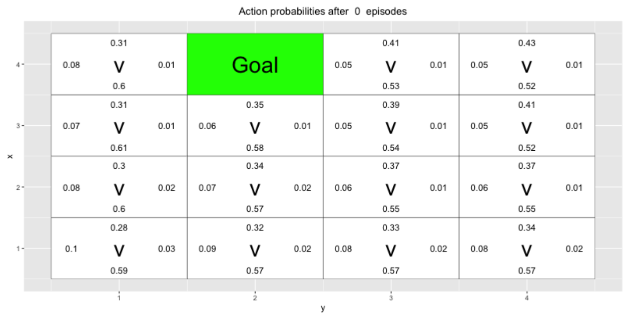

Action probabilities after 5000 episodes
Goal
x
y

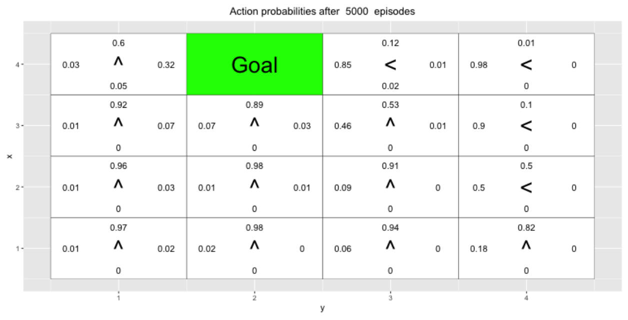

Action probabilities after 0 episodes
Goal
x
y

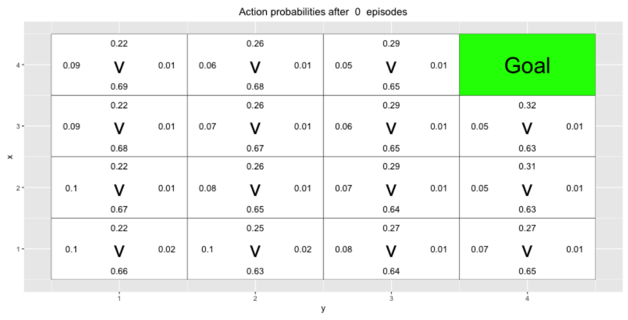

Action probabilities after 5000 episodes
Goal
x
y

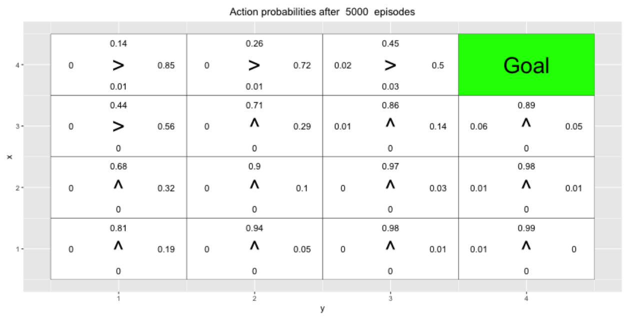

Action probabilities after 0 episodes
Goal
x
y


Action probabilities after 5000 episodes
Goal
x
y


Action probabilities after 0 episodes
Goal
x
y


Action probabilities after 5000 episodes
Goal
x
y


Action probabilities after 0 episodes
Goal
x
y


Action probabilities after 5000 episodes
Goal
x
y

- Has the agent learned a good policy? Why / Why not? Before training, the policy is distributed randomly. After 5000 episodes of training, the agent has learned a good policy, even if this was the first time that we set the goals in a different position. It has helped that the training data was diverse, the agent had goals in different parts of the square world for training, which helped in the generalization of the action-state pairs. However, it is still not the best policy because sometimes the agent failed to find the shortest path. We also tried to train the model 10000 episodes and got a better solution.

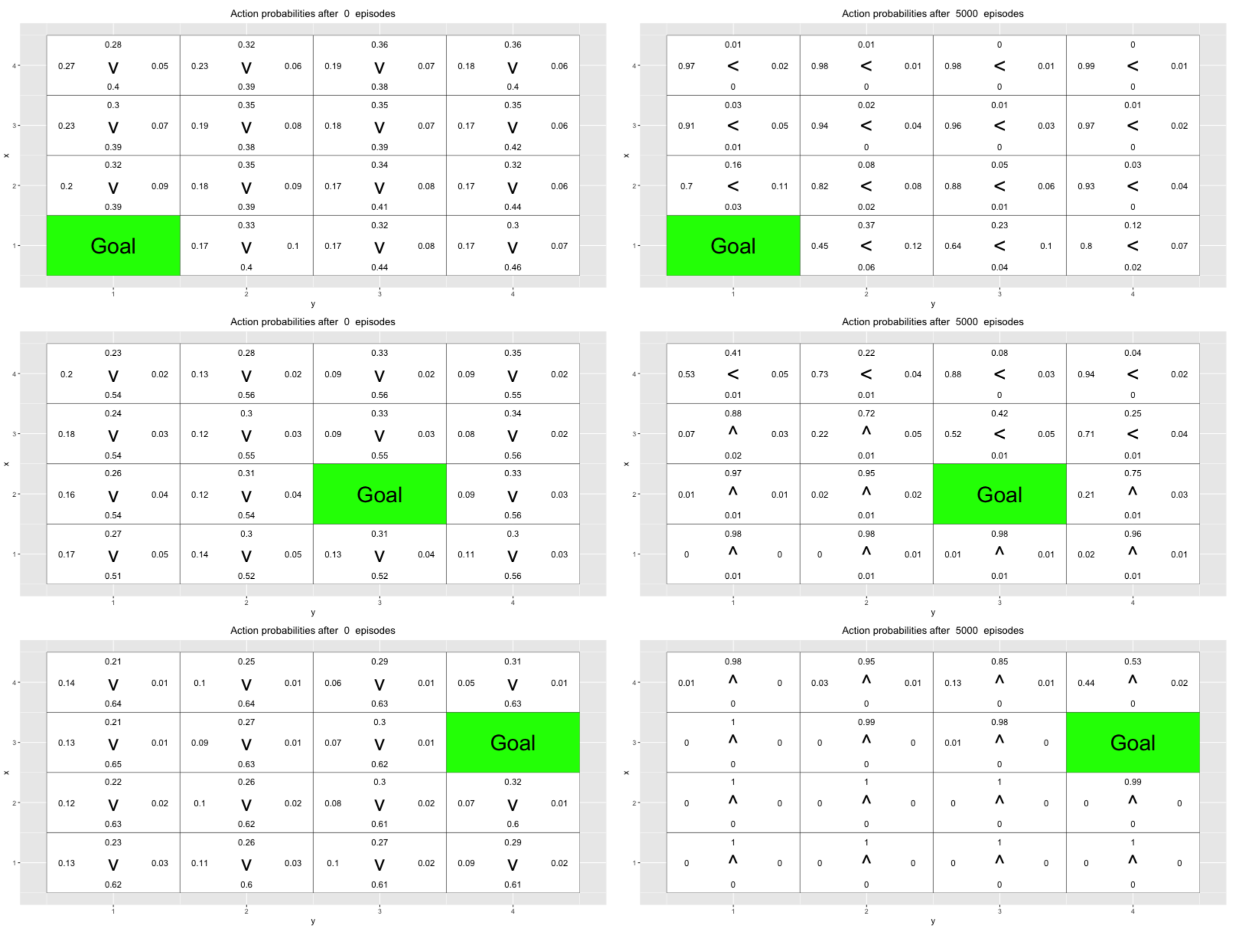
- Could you have used the Q-learning algorithm to solve this task?
We would say it is hard to solve this task by the Q-learning algorithm because in this task, the goals for training and validation are different. The Q-table is insufficient to implement it.

5.2 Environment E

You are now asked to repeat the previous experiments but this time the goals for training are all from the top row of the grid. The validation goals are three positions from the rows below. To solve this task, simply run the code provided in the file `RL_Lab2.R` and answer the following questions:

- Has the agent learned a good policy? Why / Why not ?
- If the results obtained for environments D and E differ, explain why.

Solution



The agent has not been able to learn good policies because the training and validation goals have been following slightly different patterns. The training goals are all in the upper row whereas the validation goals are more sparse. The bad results can be caused by the lack of generalization of the training environments, which is the main difference between environments D and E.

Appendix A : Given Code

```r
# 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)

# If you do not see four arrows in line 16, then do the following:
# File/Reopen with Encoding/UTF-8

#arrows <- c("↑", "→", "↓", "←")
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 = 2.5,nudge_y = .35,na.rm = TRUE) +
          geom_text(aes(label = val2),size = 2.5,nudge_x = .35,na.rm = TRUE) +
          geom_text(aes(label = val3),size = 2.5,nudge_y = -.35,na.rm = TRUE) +
          geom_text(aes(label = val4),size = 2.5,nudge_x = -.35,na.rm = TRUE) +
          geom_text(aes(label = val5),size = 5) +
          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)))

}

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)
}
```

Appendix B : Code Question 1

```r
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.

  # The coordinates
  x = start_state[1]
  y = start_state[2]
  episode_correction = 0

  repeat{
    # Follow policy, execute action, get reward.
    action = EpsilonGreedyPolicy(x, y, epsilon)
    new_state = transition_model(x, y, action, beta)
    new_x = new_state[1]
    new_y = new_state[2]
    reward = reward_map[new_x, new_y]

    # Q-table update.

    # GreedyPolicy returns the max action and handles for
    # us the multiple max cases. We won't duplicate code for
    # such purposes.
    new_maxQ = q_table[new_x, new_y, GreedyPolicy(new_x, new_y)]
    prev_maxQ = q_table[x, y, action]
    correction = reward + (gamma * new_maxQ) - prev_maxQ
    q_table[x, y, action] <<- prev_maxQ + (alpha * correction)
    episode_correction = episode_correction + correction

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

    x = new_x
    y = new_y

  }

}
```

Appendix C : Code Question 2

```
## ------------- Completing functions ------------- ##

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.

  action = which(q_table[x,y,] == max(q_table[x,y,]))
  # there could be more than one max
  # let's randomly decide which one for that case
  if (length(action) > 1) {
    action = sample(action, 1)
  }
  return(action)

}

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 greedily.
  #
  # Returns:
  #   An action, i.e. integer in {1,2,3,4}.

  # Your code here.

  # If the random number is higher than the eps-threshold
  # then explore the environment. If not, exploit the environment
  if (epsilon > runif(1)) {
    action = GreedyPolicy(x, y)
  } else {
    action = sample(1:4, 1)
  }

  return(action)

}
## ------------- Testbench ------------- ##
```

```r
# 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()

set.seed(12345)
for(i in 1:10000){
  foo <- q_learning(start_state = c(3,1))

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

Appendix D : Code Question 3

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

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")
}
```

Appendix E : Code Question 4

```
# 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)
}
```

Appendix F : REINFORCE : Given code

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

#####################################################################################################
# REINFORCE
#####################################################################################################

# install.packages("keras")
library(keras)

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

# If you do not see four arrows in line 19, then do the following:
# File/Reopen with Encoding/UTF-8

arrows <- c("↑", "→", "↓", "←")
action_deltas <- list(c(1,0), # up
                      c(0,1), # right
                      c(-1,0), # down
                      c(0,-1)) # left

vis_prob <- function(goal, episodes = 0){

  # Visualize an environment with rewards.
  # Probabilities for all actions are displayed on the edges of each tile.
  # The (greedy) policy for each state is also displayed.
  #
  # Args:
  #   goal: goal coordinates, array with 2 entries.
  #   episodes, epsilon, alpha, gamma, beta (optional): for the figure title.
  #   H, W (global variables): environment dimensions.

  df <- expand.grid(x=1:H,y=1:W)
  dist <- array(data = NA, dim = c(H,W,4))
  class <- array(data = NA, dim = c(H,W))
  for(i in 1:H)
    for(j in 1:W){
      dist[i,j,] <- DeepPolicy_dist(i,j,goal[1],goal[2])
      foo <- which(dist[i,j,]==max(dist[i,j,]))
      class[i,j] <- ifelse(length(foo)>1,sample(foo, size = 1),foo)
    }

  foo <- mapply(function(x,y) ifelse(all(c(x,y) == goal),NA,dist[x,y,1]),df$x,df$y)
  df$val1 <- as.vector(round(foo, 2))
  foo <- mapply(function(x,y) ifelse(all(c(x,y) == goal),NA,dist[x,y,2]),df$x,df$y)
  df$val2 <- as.vector(round(foo, 2))
  foo <- mapply(function(x,y) ifelse(all(c(x,y) == goal),NA,dist[x,y,3]),df$x,df$y)
  df$val3 <- as.vector(round(foo, 2))
```

```r
  foo <- mapply(function(x,y) ifelse(all(c(x,y) == goal),NA,dist[x,y,4]),df$x,df$y)
  df$val4 <- as.vector(round(foo, 2))
  foo <- mapply(function(x,y) ifelse(all(c(x,y) == goal),NA,class[x,y]),df$x,df$y)
  df$val5 <- as.vector(arrows[foo])
  foo <- mapply(function(x,y) ifelse(all(c(x,y) == goal),"Goal",NA),df$x,df$y)
  df$val6 <- as.vector(foo)

  print(ggplot(df,aes(x = y,y = x)) +
          geom_tile(fill = 'white', colour = 'black') +
          scale_fill_manual(values = c('green')) +
          geom_tile(aes(fill=val6), show.legend = FALSE, colour = 'black') +
          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,na.rm = TRUE) +
          geom_text(aes(label = val6),size = 10,na.rm = TRUE) +
          ggtitle(paste("Action probabilities after ",episodes," episodes")) +
          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)))

}

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)
}

DeepPolicy_dist <- function(x, y, goal_x, goal_y){

  # Get distribution over actions for state (x,y) and goal (goal_x,goal_y) from the deep policy.
  #
  # Args:
  #   x, y: state coordinates.
  #   goal_x, goal_y: goal coordinates.
  #   model (global variable): NN encoding the policy.
  #
  # Returns:
  #   A distribution over actions.

  foo <- matrix(data = c(x,y,goal_x,goal_y), nrow = 1)

  # return (predict_proba(model, x = foo))
  return (predict_on_batch(model, x = foo)) # Faster.

}

DeepPolicy <- function(x, y, goal_x, goal_y){

  # Get an action for state (x,y) and goal (goal_x,goal_y) from the deep policy.
  #
  # Args:
  #   x, y: state coordinates.
  #   goal_x, goal_y: goal coordinates.
  #   model (global variable): NN encoding the policy.
  #
  # Returns:
  #   An action, i.e. integer in {1,2,3,4}.

  foo <- DeepPolicy_dist(x,y,goal_x,goal_y)

  return (sample(1:4, size = 1, prob = foo))

}

DeepPolicy_train <- function(states, actions, goal, gamma){

  # Train the policy network on a rolled out trajectory.
  #
  # Args:
  #   states: array of states visited throughout the trajectory.
  #   actions: array of actions taken throughout the trajectory.
  #   goal: goal coordinates, array with 2 entries.
  #   gamma: discount factor.

  # Construct batch for training.
  inputs <- matrix(data = states, ncol = 2, byrow = TRUE)
  inputs <- cbind(inputs,rep(goal[1],nrow(inputs)))
  inputs <- cbind(inputs,rep(goal[2],nrow(inputs)))

  targets <- array(data = actions, dim = nrow(inputs))
  targets <- to_categorical(targets-1, num_classes = 4)

  # Sample weights. Reward of 5 for reaching the goal.
  weights <- array(data = 5*(gamma^(nrow(inputs)-1)), dim = nrow(inputs))

  # Train on batch. Note that this runs a SINGLE gradient update.
  train_on_batch(model, x = inputs, y = targets, sample_weight = weights)
```

```r
}

reinforce_episode <- function(goal, gamma = 0.95, beta = 0){
  
  # Rolls out a trajectory in the environment until the goal is reached.
  # Then trains the policy using the collected states, actions and rewards.
  # 
  # Args:
  #   goal: goal coordinates, array with 2 entries.
  #   gamma (optional): discount factor.
  #   beta (optional): probability of slipping in the transition model.
  
  # Randomize starting position.
  cur_pos <- goal
  while(all(cur_pos == goal))
    cur_pos <- c(sample(1:H, size = 1),sample(1:W, size = 1))
  
  states <- NULL
  actions <- NULL
  
  steps <- 0 # To avoid getting stuck and/or training on unnecessarily long episodes.
  while(steps < 20){
    steps <- steps+1
    
    # Follow policy and execute action.
    action <- DeepPolicy(cur_pos[1], cur_pos[2], goal[1], goal[2])
    new_pos <- transition_model(cur_pos[1], cur_pos[2], action, beta)
    
    # Store states and actions.
    states <- c(states,cur_pos)
    actions <- c(actions,action)
    cur_pos <- new_pos
    
    if(all(new_pos == goal)){
      # Train network.
      DeepPolicy_train(states,actions,goal,gamma)
      break
    }
  }
  
}

####################################################################################################
# REINFORCE Environments
####################################################################################################

# Environment D (training with random goal positions)

H <- 4
W <- 4

# Define the neural network (two hidden layers of 32 units each).
model <- keras_model_sequential()
model %>%
  layer_dense(units = 32, input_shape = c(4), activation = 'relu') %>%
  layer_dense(units = 32, activation = 'relu') %>%
  layer_dense(units = 4, activation = 'softmax')

compile(model, loss = "categorical_crossentropy", optimizer = optimizer_sgd(lr=0.001))

initial_weights <- get_weights(model)

train_goals <- list(c(4,1), c(4,3), c(3,1), c(3,4), c(2,1), c(2,2), c(1,2), c(1,3))
val_goals <- list(c(4,2), c(4,4), c(3,2), c(3,3), c(2,3), c(2,4), c(1,1), c(1,4))

show_validation <- function(episodes, env){

  for(goal in val_goals) {
    png(filename = paste("env", env, "_goal_", paste(goal, collapse = ""),
                         "episodes", episodes,
                         ".png", sep = ""),
        width=800, height=400)
    vis_prob(goal, episodes)
    dev.off()
  }

}

set_weights(model,initial_weights)

show_validation(0, "D")

for(i in 1:5000){
  if(i%%10==0) cat("episode",i,"\n")
  goal <- sample(train_goals, size = 1)
  reinforce_episode(unlist(goal))
}

show_validation(5000, "D")

# Environment E (training with top row goal positions)

train_goals <- list(c(4,1), c(4,2), c(4,3), c(4,4))
val_goals <- list(c(3,4), c(2,3), c(1,1))

set_weights(model,initial_weights)

show_validation(0, "E")

for(i in 1:5000){
  if(i%%10==0) cat("episode", i,"\n")
  goal <- sample(train_goals, size = 1)
  reinforce_episode(unlist(goal))
}

show_validation(5000, "E")
```

Appendix G : REINFORCE : Environment D

```r
knitr::include_graphics(c("src/envD_goal_11episodes0.png","src/envD_goal_11episodes5000.png"))
knitr::include_graphics(c("src/envD_goal_14episodes0.png","src/envD_goal_14episodes5000.png"))
knitr::include_graphics(c("src/envD_goal_23episodes0.png","src/envD_goal_23episodes5000.png"))
knitr::include_graphics(c("src/envD_goal_24episodes0.png","src/envD_goal_24episodes5000.png"))
knitr::include_graphics(c("src/envD_goal_32episodes0.png","src/envD_goal_32episodes5000.png"))
knitr::include_graphics(c("src/envD_goal_33episodes0.png","src/envD_goal_33episodes5000.png"))
knitr::include_graphics(c("src/envD_goal_42episodes0.png","src/envD_goal_42episodes5000.png"))
knitr::include_graphics(c("src/envD_goal_44episodes0.png","src/envD_goal_44episodes5000.png"))
```

Appendix G : REINFORCE : Environment E

```r
knitr::include_graphics(c("src/envE_goal_11episodes0.png","src/envE_goal_11episodes5000.png"))
knitr::include_graphics(c("src/envE_goal_23episodes0.png","src/envE_goal_23episodes5000.png"))
knitr::include_graphics(c("src/envE_goal_34episodes0.png","src/envE_goal_34episodes5000.png"))
```

References

[1] J. M. Peña and J. Oskarsson, Linköpings Universitet, 2020.