LAB 3: REINFORCEMENT LEARNING

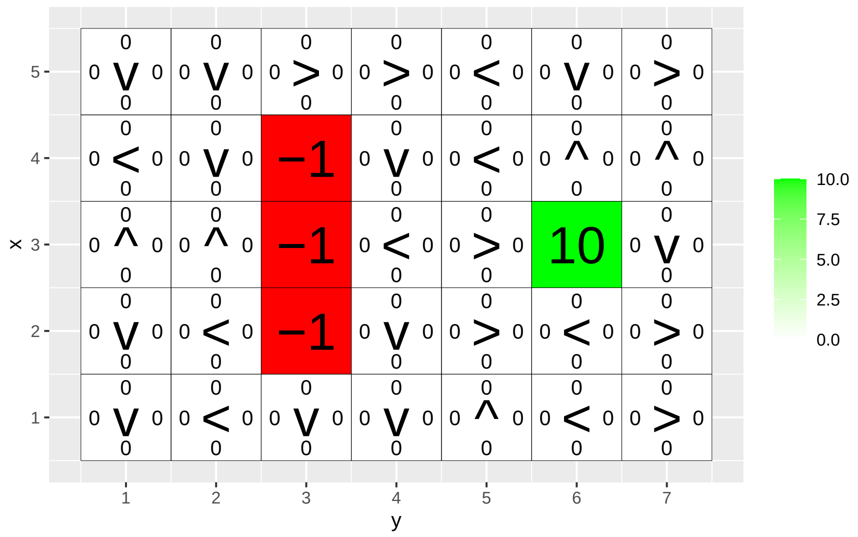
Arash Haratian, Daniel Díaz-Roncero González, Elena Dalla Torre & Juan Manuel Pardo Ladino

Environment A

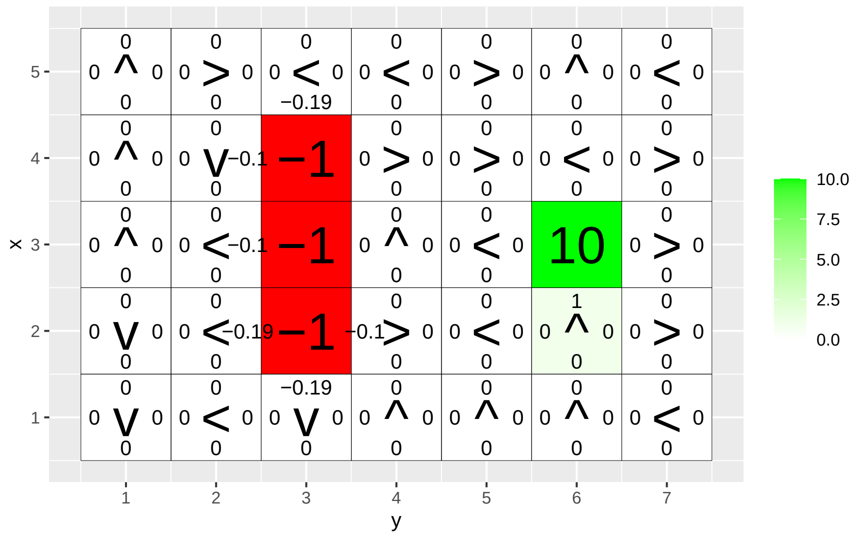
```r
# By Jose M. Peña and Joel Oskarsson.
# For teaching purposes.
# jose.m.pena@liu.se.

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

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

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.
  action <- which.is.max(q_table[x,y,])
  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 randomly.
  #
  # Returns:
  #   An action, i.e. integer in {1,2,3,4}.

  # Your code here.
  u <- runif(1,0,1)
  if(u < epsilon){
```

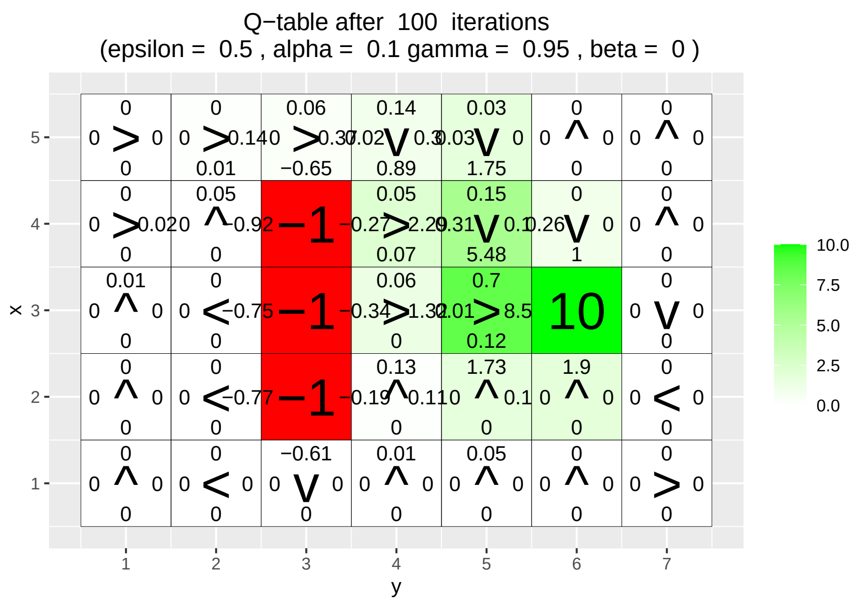
```r
    action <- floor(runif(1,1,5))
  }else {
    action <- GreedyPolicy(x,y)
  }
  return(action)
}

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 randomly.
  #   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
  repeat{
    # Follow policy, execute action, get reward.
    action <- EpsilonGreedyPolicy(x,y,epsilon)
    newstate <- transition_model(x, y, action, beta)
    ns_x <- newstate[1]
    ns_y <- newstate[2]
    reward <- reward_map[ns_x,ns_y]

    # Q-table update.
    correction <- reward + gamma * max(q_table[ns_x,ns_y,]) - q_table[x,y,action]
    episode_correction <- episode_correction + correction
    q_table[x,y,action] <<- q_table[x,y,action] + alpha*(correction)
    x <- ns_x
    y <- ns_y

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

}

#####################################################################################################
# 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()

set.seed(111)
for(i in 1:10000){
  foo <- q_learning(start_state = c(3,1))

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

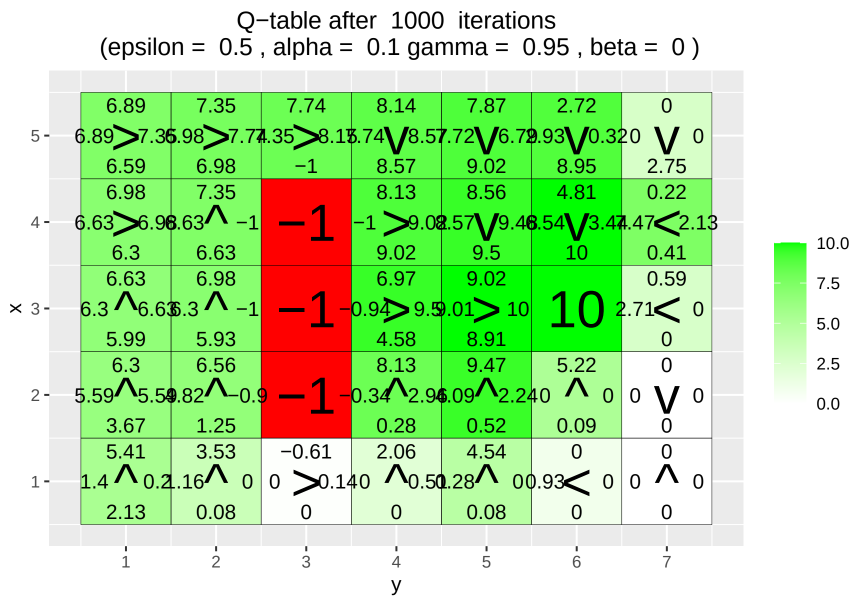
Q−table after 0 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.95 , beta = 0)

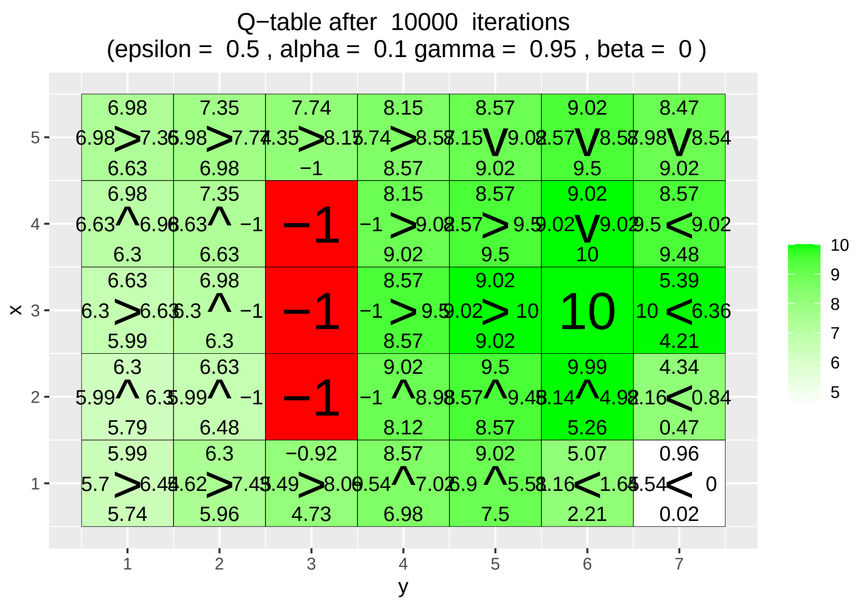


Q−table after 10 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.95 , beta = 0)




Q−table after 100 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.95 , beta = 0)
x
y
10.0
7.5
5.0
2.5
0.0


Q−table after 1000 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.95 , beta = 0)
x
y
10.0
7.5
5.0
2.5
0.0



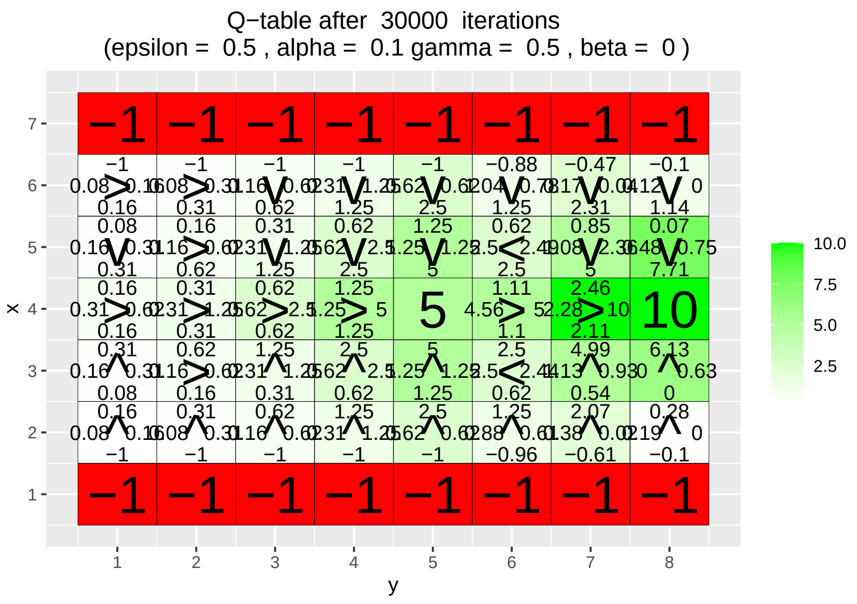
After the first 10 iterations the agent hasn't reached yet the good reward, but it has reached the negative rewards from 6 different states (depending on the run, the number of these states might change) and it has learned that it shouldn't go directly to that state from these states.

It's not optimal for all states, for example if we look at $(x = 1, y = 6)$ we see that the greedy policy says to go to the left, when actually the "best" (fastest) path to the reward would be to go up.

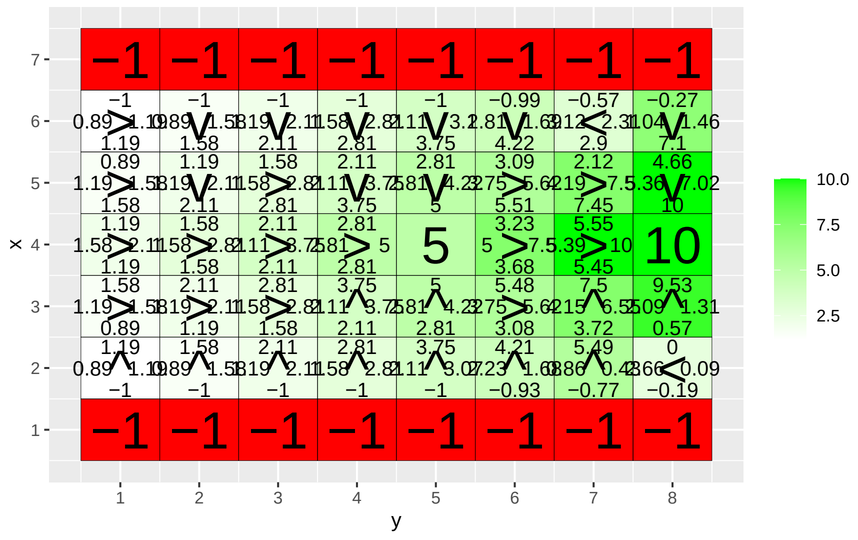
Yes, the values from the Q-table reflect both paths, under and over the negative rewards, but this is random and depends on the run.

Environment B

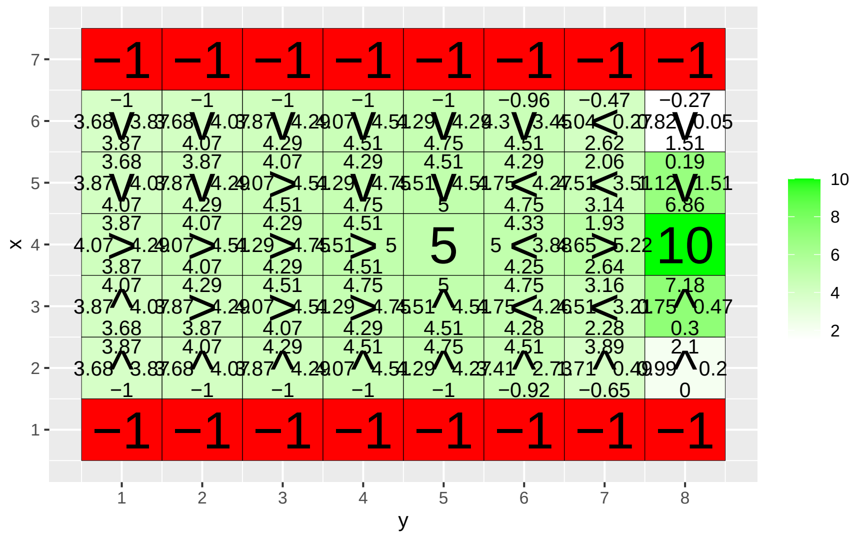
```r
# Environment B (the effect of epsilon and gamma)
set.seed(222)
H <- 7
W <- 8

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")
}
```



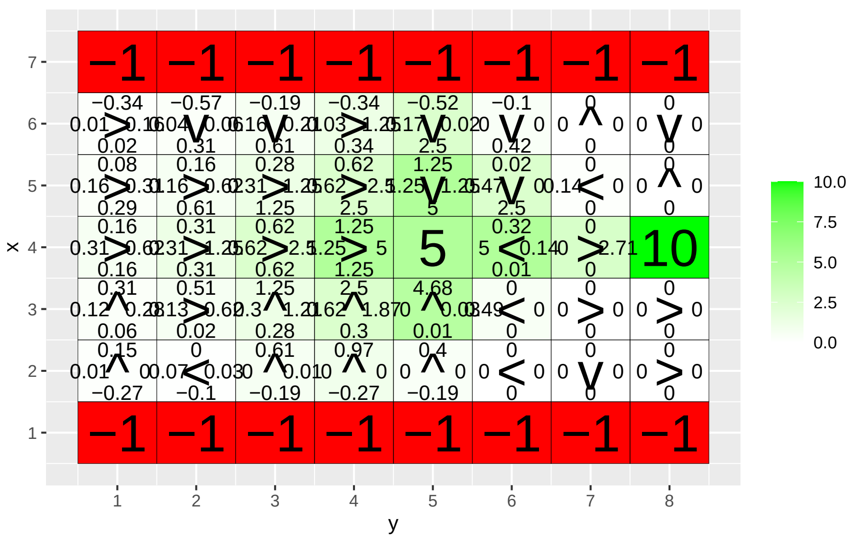
Q−table after 30000 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.75 , beta = 0)



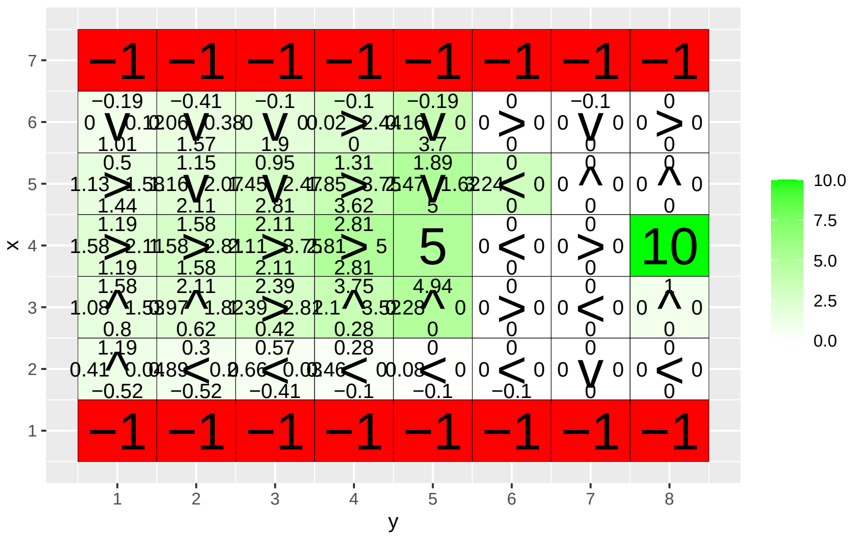
Q−table after 30000 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.95 , beta = 0)

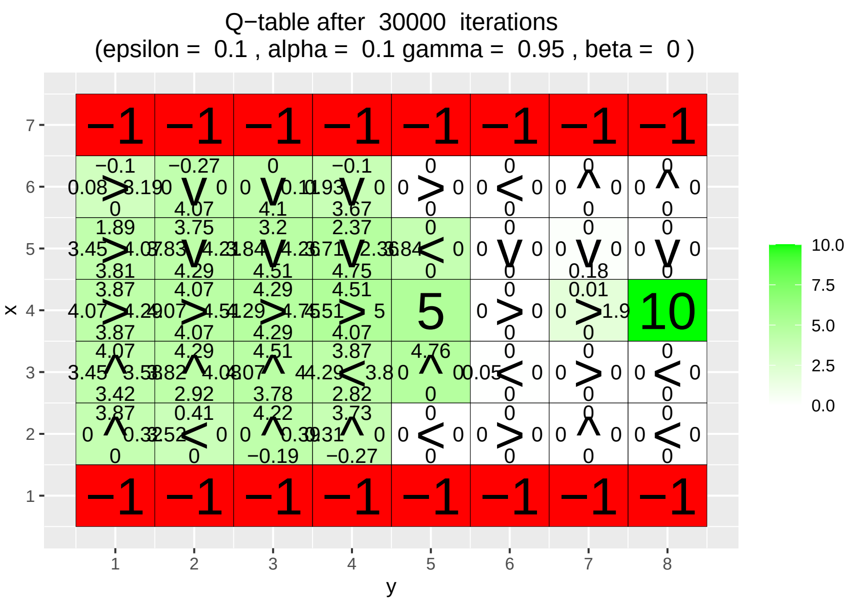


Q−table after 30000 iterations
(epsilon = 0.1 , alpha = 0.1 gamma = 0.5 , beta = 0)



Q−table after 30000 iterations
(epsilon = 0.1 , alpha = 0.1 gamma = 0.75 , beta = 0)





Let’s analyze first the different cases:

- When epsilon = 0.5, alpha = 0.1, gamma = 0.5, beta = 0: The agent finds both rewards and the policy dictates to go to that one which is closer.
- When epsilon = 0.5, alpha = 0.1, gamma = 0.75, beta = 0: The agent finds both rewards and when it is in a state between the two positive rewards, the policy dictates to go to the highest reward (the 10).
- When epsilon = 0.5, alpha = 0.1, gamma = 0.95, beta = 0: The agent finds both rewards and the policy deliberately avoids the “5” reward to go to the highest reward (the 10).
- When epsilon = 0.1, alpha = 0.1, gamma = 0.5, beta = 0: The agent doesn’t find the “10” reward and the policy dictates to go to the “5” reward in the fastest way.
- When epsilon = 0.1, alpha = 0.1, gamma = 0.75, beta = 0: We get very similar results as with gamma 0.5, but this time because of the randomness of the process it reached the “10” policy once and for that state knows how to go to the “10” reward.
- When epsilon = 0.1, alpha = 0.1, gamma = 0.95, beta = 0: The agent doesn’t find the “10” reward and the policy tries to follow the initial paths that it found as with the high gamma, the values of the Q-table are very high from the beginning.

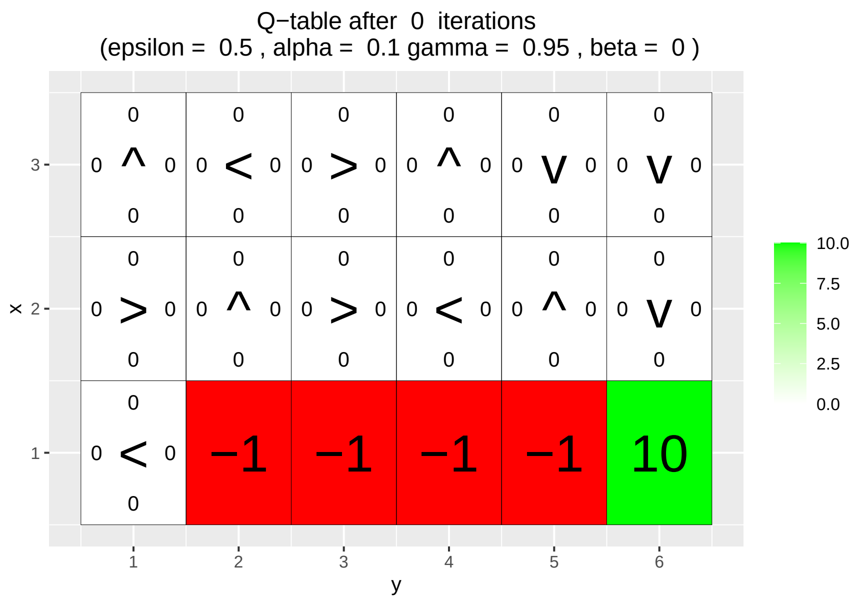
From these we can reach the following conclusions:

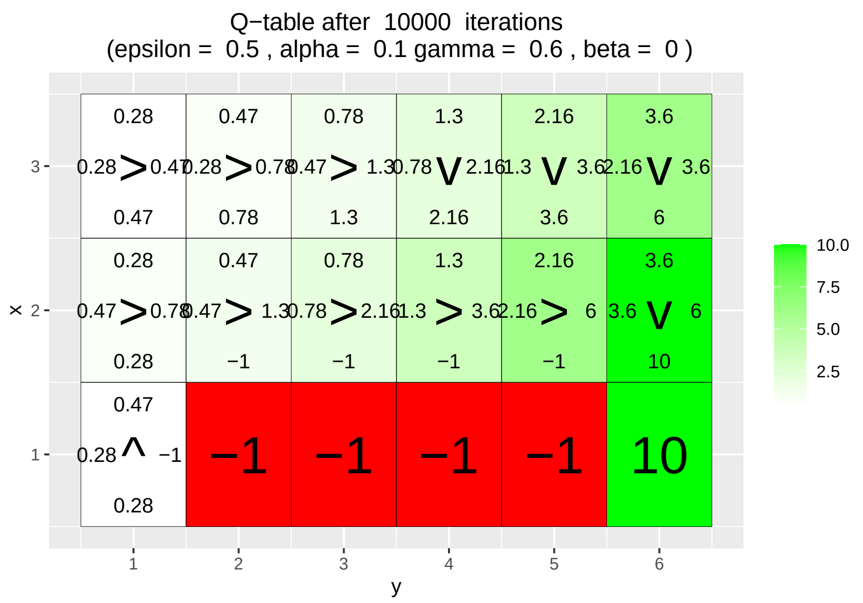
- A higher epsilon allows for the agent to find the different positive rewards, as if even when one reward has been found, the epsilon greedy policy allows for the agent to deviate from the known positive paths to explore the grid and find other rewards.
- A higher gamma value with a reasonably high epsilon value will make the policy to dictate that the agent goes to the highest reward (even when a lower positive reward is close), as the epsilon is high enough for the agent to find the highest reward and the gamma makes much more valuable this reward, so once it is found, the instruction to get to this reward will propagate through the grid easier and faster.
- A lower epsilon will make the agent to not deviate much from the initial paths that reach a positive result, which impedes the agent to find other positive results, even when they are higher.

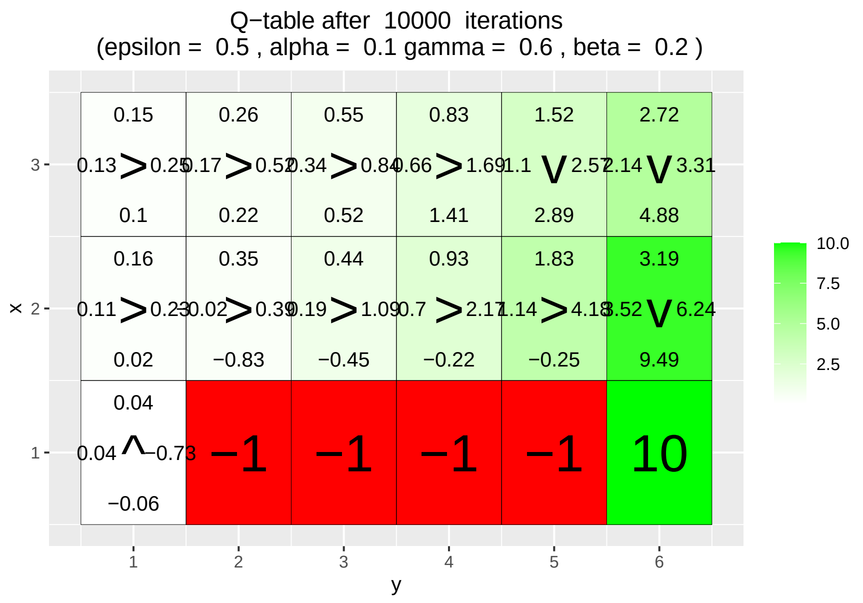
- A lower gamma value with a low epsilon will make the agent to not deviate at all from the first paths that it finds to the reward, even if that means to reach the reward faster, as the Q-values for the states for these initials paths will be very high and the low epsilon doesn't allow to try many more different paths.

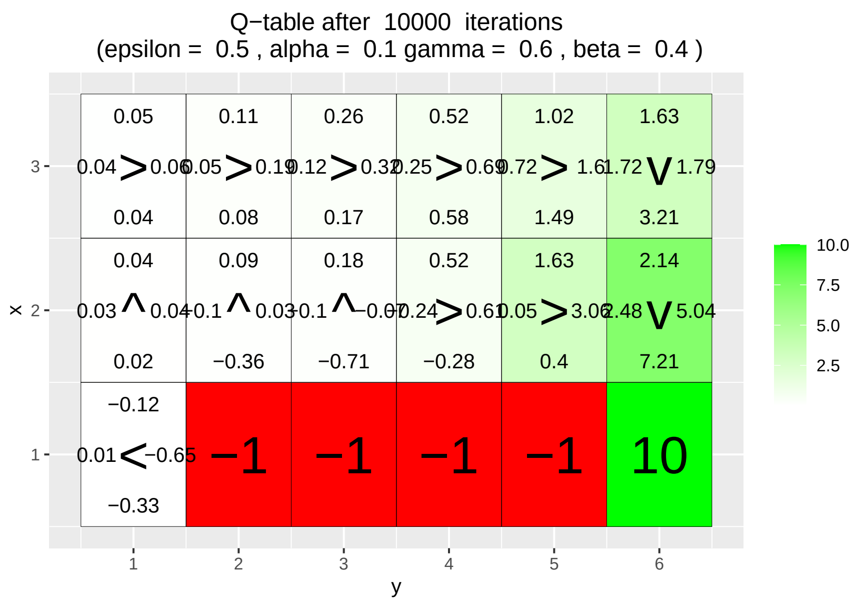
Environment C

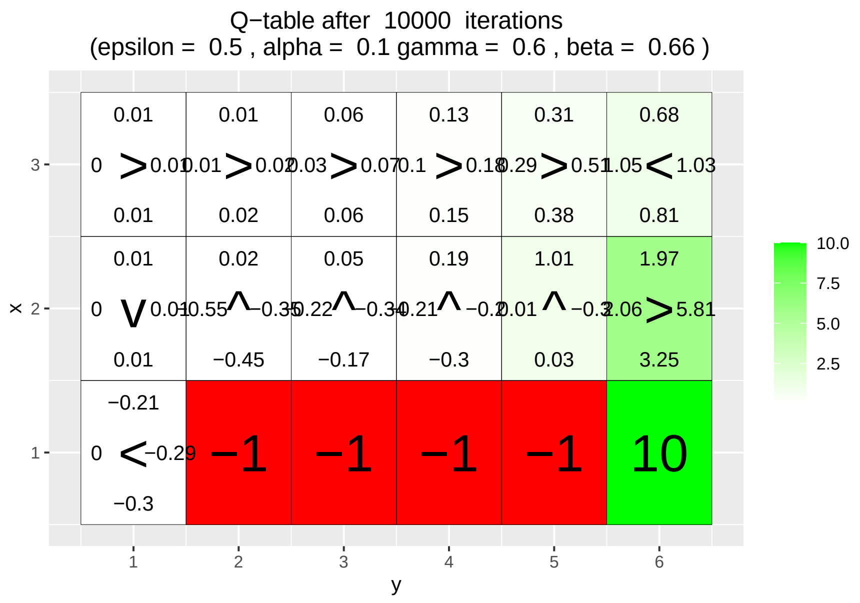
```r
# Environment C (the effect of beta).

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)
}
```




Q−table after 10000 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.6 , beta = 0)
0.28 0.47 0.78 1.3 2.16 3.6
0.28 > 0.47 0.28 > 0.78 0.47 > 1.3 0.78 V 2.16 1.3 V 3.6 2.16 V 3.6
0.47 0.78 1.3 2.16 3.6 6
0.28 0.47 0.78 1.3 2.16 3.6
0.47 > 0.78 0.47 > 1.3 0.78 > 2.16 1.3 > 3.6 2.16 > 6 3.6 V 6
0.28 −1 −1 −1 −1 10
0.47
0.28 ^ −1 −1 −1 −1 −1 10
0.28
x
3
2
1
y
1 2 3 4 5 6
10.0
7.5
5.0
2.5


Q−table after 10000 iterations
(epsilon = 0.5 , alpha = 0.1 gamma = 0.6 , beta = 0.2)
0.15 0.26 0.55 0.83 1.52 2.72
0.13 > 0.25 0.17 > 0.52 0.34 > 0.84 0.66 > 1.69 1.1 V 2.57 2.14 V 3.31
0.1 0.22 0.52 1.41 2.89 4.88
0.16 0.35 0.44 0.93 1.83 3.19
0.11 > 0.23 −0.02 > 0.39 0.19 > 1.09 0.7 > 2.17 1.14 > 4.18 3.52 V 6.24
0.02 −0.83 −0.45 −0.22 −0.25 9.49
0.04
0.04 ^ −0.73 −1 −1 −1 −1 10
−0.06
x
3
2
1
y
1 2 3 4 5 6
10.0
7.5
5.0
2.5





When the beta is 0, there is no possibility of slipping to the right or left of the direction dictated by the policy (the agent can still go to different states because of the epsilon, but no possibility of slipping), hence the policy dictates to go to the positive reward in the fastest way. As the beta starts to grow, the policy tries to avoid the possibility of falling into the negative rewards, starting from the states of the top row (x=3) and the states of the first columns of the middle row (x=2), as if in these states of the second row if it kept going to the right instead of going up, it would have chances to fall into the negative rewards and in the second row the farther the agent is from the positive reward (the first columns) the more chances it has to fall to the negative rewards as it has more states with negative rewards to the right to go through. For the beta = 0.66 we now see that the policy dictates that for all values (except for y=6) of the second row, the agent should go up, it is also noticeable to see that for x=1,y=1 the policy says to fo to the left, to have 0 chances of falling to the negative reward in x=1,y=2.

As beta goes up the reward that we get is more noisy and the policy might not be the fastest. For beta equal to 0.66 the policy tries to avoid the possibility of falling into negative rewards and ends up dictating to go to a cell which might be in the opposite direction of the reward and might create some loops.

Environment D

The code used for this task is:

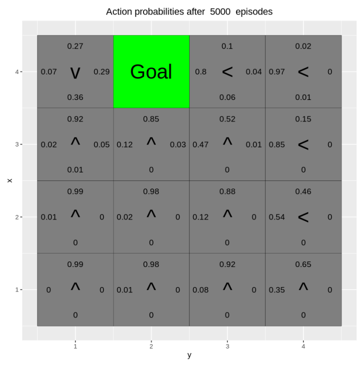
```r
#####################################################################################################
# REINFORCE
#####################################################################################################

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

# 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_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$val3 <- as.vector(round(foo, 2))
  foo <- mapply(function(x,y) ifelse(all(c(x,y) == goal),NA,dist[x,y,4]),df$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)
```

```r
  # 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)

}

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))
```

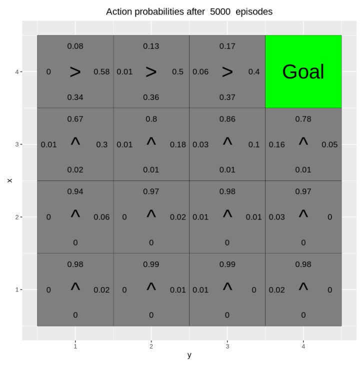
```r
initial_weights <- get_weights(model)

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){

  for(goal in val_goals)
    vis_prob(goal, episodes)

}

set_weights(model,initial_weights)

show_validation(0)

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)
```

The finals plots (after 5000 episodes) from this environment are the following, (we had the images saved and now we load them):

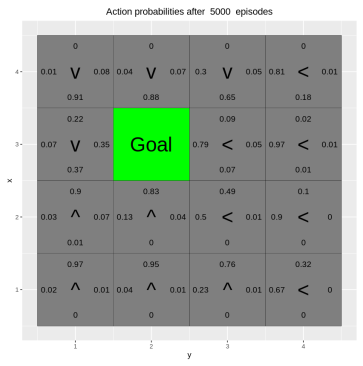
```r
library(imager)
plot(load.image("ENVd_val1.png"),axes=FALSE)
plot(load.image("ENVd_val2.png"),axes=FALSE)
plot(load.image("ENVd_val3.png"),axes=FALSE)
plot(load.image("ENVd_val4.png"),axes=FALSE)
plot(load.image("ENVd_val5.png"),axes=FALSE)
plot(load.image("ENVd_val6.png"),axes=FALSE)
plot(load.image("ENVd_val7.png"),axes=FALSE)
plot(load.image("ENVd_val8.png"),axes=FALSE)
```

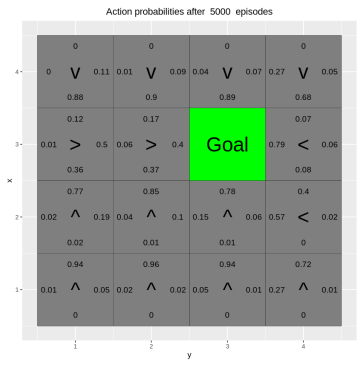


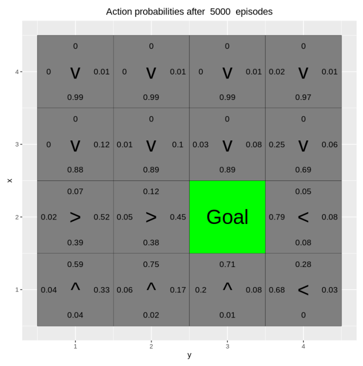
Action probabilities after 5000 episodes
0.27
0.07 v 0.29
0.36
Goal
0.1
0.8 < 0.04
0.06
0.02
0.97 < 0
0.01
0.92
0.02 ^ 0.05
0.01
0.85
0.12 ^ 0.03
0
0.52
0.47 ^ 0.01
0
0.15
0.85 < 0
0
0.99
0.01 ^ 0
0
0.98
0.02 ^ 0
0
0.88
0.12 ^ 0
0
0.46
0.54 < 0
0
0.99
0 ^ 0
0
0.98
0.01 ^ 0
0
0.92
0.08 ^ 0
0
0.65
0.35 ^ 0
0
x
4
3
2
1
y
1
2
3
4

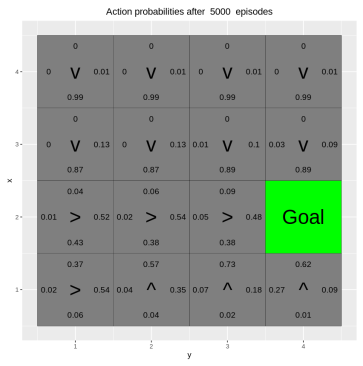


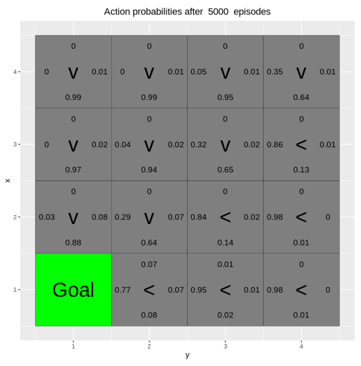
Action probabilities after 5000 episodes
0.08
0 > 0.58
0.34
0.13
0.01 > 0.5
0.36
0.17
0.06 > 0.4
0.37
Goal
0.67
0.01 ^ 0.3
0.02
0.8
0.01 ^ 0.18
0.01
0.86
0.03 ^ 0.1
0.01
0.78
0.16 ^ 0.05
0.01
0.94
0 ^ 0.06
0
0.97
0 ^ 0.02
0
0.98
0.01 ^ 0.01
0
0.97
0.03 ^ 0
0
0.98
0 ^ 0.02
0
0.99
0 ^ 0.01
0
0.99
0.01 ^ 0
0
0.98
0.02 ^ 0
0
x
4
3
2
1
y
1
2
3
4

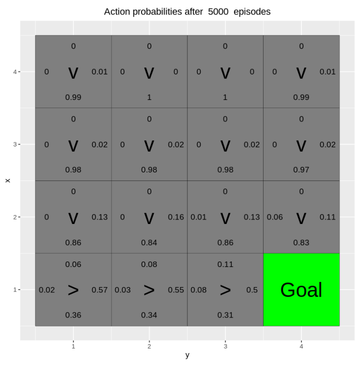

Action probabilities after 5000 episodes
Goal
x
y


Action probabilities after 5000 episodes
Goal
x
y


Action probabilities after 5000 episodes
0 0 V 0.01 0.99
0 0 V 0.01 0.99
0 0 V 0.01 0.99
0 0.02 V 0.01 0.97
0 0 V 0.12 0.88
0 0.01 V 0.1 0.89
0 0.03 V 0.08 0.89
0 0.25 V 0.06 0.69
0.07 0.02 > 0.52 0.39
0.12 0.05 > 0.45 0.38
Goal
0.05 0.79 < 0.08 0.08
0.59 0.04 ^ 0.33 0.04
0.75 0.06 ^ 0.17 0.02
0.71 0.2 ^ 0.08 0.01
0.28 0.68 < 0.03 0
x
4
3
2
1
y
1
2
3
4


Action probabilities after 5000 episodes
0 0 V 0.01 0.99
0 0 V 0.01 0.99
0 0 V 0.01 0.99
0 0 V 0.01 0.99
0 0 V 0.13 0.87
0 0 V 0.13 0.87
0 0.01 V 0.1 0.89
0 0.03 V 0.09 0.89
0.04 0.01 > 0.52 0.43
0.06 0.02 > 0.54 0.38
0.09 0.05 > 0.48 0.38
Goal
0.37 0.02 > 0.54 0.06
0.57 0.04 ^ 0.35 0.04
0.73 0.07 ^ 0.18 0.02
0.62 0.27 ^ 0.09 0.01
x
4
3
2
1
y
1
2
3
4





In general the agent does learn a good policy that reaches the reward in the fastest way, although for some specific states, the policy might not be optimal.

No, the Q-learning algorithm could not have been used as you can not train on some goals and then use some other goals for validation. Additionally, the Q algorithm would not work well in this case because it's policy is deterministic and does not take any additional parameters. This means that it can only generate a policy for a fixed reward position.The Q Learning learns a parameterized Q table (state-action value table) based on the given environment. If the environment and the location of the goal changes, the learnt Q-values are not correct anymore, and agent cannot act based on the values and the policy derived from it.

The reinforcement algorithm suits perfectly this problem because we need the reward location as a parameter and this algorithm is indeed a parametrized algorithm. Another reason why it works well in this case is because the training data varied the location of the reward over the entire grid, and therefore the agent can raise a general space awareness to be able to decide the proper direction to go depending on the reward location and initial state.

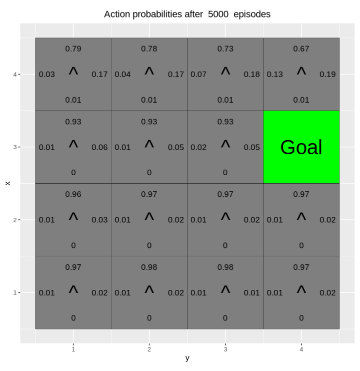
Environment E

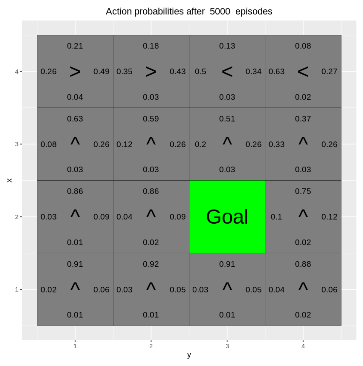
The code for this task is:

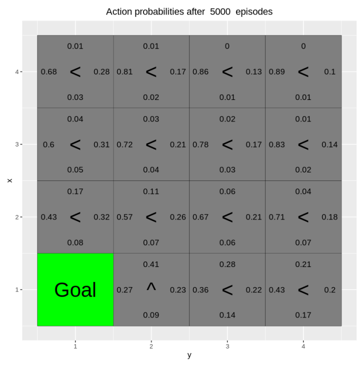
```r
# Environment E (training with top row goal positions)

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)

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)
```

The finals plots (after 5000 episodes) from this environment are the following, (we had the images saved and now we load them):

```r
library(imager)

plot(load.image("ENVe_val1.png"),axes=FALSE)
plot(load.image("ENVe_val2.png"),axes=FALSE)
plot(load.image("ENVe_val3.png"),axes=FALSE)
```







The agent hasn't learned a good policy, it seems that since the training goals were all in the top row, we have an overfitting problem were the agent has learned to always go in the same direction, which in two of the three validation cases is to go up, which makes sense having into account that for the training data they always needed to go up and then to specific state in the top row. In the last validation case, the policy dictates to always go to the left (excepting state x=1,y=2) which isn't either a good policy, since it's not telling the agent to go to the goal, and when it reaches the first column, it is not told to go down. The reason behind this behaviour might be that, for the first two validation cases, the agent knows that at some point it needs to go up (since it knows where the goal is and for some states it in fact needs to go up) and since it has learned from training to go up until the first row, then it does so, while for the third case it knows that for no state it needs to go up (since it knows the position of the goal and that is the first row) and from the training data (we had the (4,1) training data) it learned that if the goal is in the first column at some point it would need to go full left, hence it dictates to only go full left.

As explained the results differ because of the overfitting. The way the environment changes now is different. In fact, the policy network classify most of the states to left action as the sample weights for those state-action pairs are big. But, when the goal moves to somewhere else on the map, the network would not generate a better action as it did not get enough state-action pairs with actions heading towards the new goal with big sample weights; in fact, the sample weight probably for any actions except left were low. However, in question D, the agent interacts with the training environment that is more close to the validation. In other words, in environment E the trajectories are biased in compare to the validation maps but in environment D the trajectories are unbiased.

Statement of Contribution

All the members of the group contributed with code, text and discussions in every question. The individuals solutions of all members were compared in order to decide the best approach. Although the four students contributed to every question a more detailed list of who contributed more to each question would be the following:

- Enviroment A: Daniel Díaz-Roncero González and Juan Manuel Pardo Ladino
- Enviroment B: Elena Dalla Torre and Arash Haratian
- Enviroment C: Juan Manuel Pardo Ladino and Elena Dalla Torre
- Enviroment D: All contributed equally
- Enviroment E: Arash Haratian and Daniel Díaz-Roncero González